

Processus compositionnels interactifs : une architecture pour la programmation et l'exécution des structures musicales.

Dimitri Bouche

▶ **To cite this version:**

Dimitri Bouche. Processus compositionnels interactifs : une architecture pour la programmation et l'exécution des structures musicales.. Ingénierie assistée par ordinateur. Université Pierre et Marie Curie - Paris VI, 2016. Français. NNT : 2016PA066533 . tel-01524393

HAL Id: tel-01524393
https://theses.hal.science/tel-01524393

Submitted on 18 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Dimitri BOUCHE

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Processus compositionnels interactifs : Une architecture pour la programmation et l'exécution des structures musicales

Soutenue le 12 décembre 2016 devant le jury composé de :

Gérard Assayag	Ircam UMR STMS (Directeur)
Jean Bresson	Ircam UMR STMS (Encadrant)
Jean-Pierre Briot	Université Pierre et Marie Curie
Elaine Chew	Queen Mary, University of London
Roger Dannenberg	Carnegie Mellon University
Philippe Manoury	Collège de France (Invité)
Yann Orlarey	Grame
Peter Van Roy	Université Catholique de Louvain

Rapporteurs :

Miller Puckette	University of California San Diego
Peter Van Roy	Université Catholique de Louvain

Dimitri Bouche — *Processus compositionnels interactifs : Une architecture pour la programmation et l'exécution des structures musicales*

ENCADRANT :
Jean Bresson
DIRECTEUR :
Gérard Assayag

Cette querelle du « tout-temps-réel » ou du « tout-temps-différé » me semble pouvoir être définitivement levée car, comme dans tous les cas semblables, ce ne sont jamais les positions extrêmes qui aboutissent aux résultats les plus intéressants.

Philippe Manoury, « La querelle des temps »

RÉSUMÉ

Les outils de composition musicale assistée par ordinateur (CAO) permettent la génération de données musicales à partir de programmes informatiques conçus par un utilisateur. Ces programmes sont qualifiés de « temps-différés » pour dénoter la séparation entre les temps inclus dans leurs résultats, et le temps absolu qui s'écoule lors de leurs exécutions. Les outils de CAO offrent une représentation temporelle des données musicales, permettant ainsi de composer à l'échelle de la structure de l'œuvre globale. En revanche, ils ne permettent pas d'interagir avec cette structure pendant sa restitution. Cela contraint l'utilisateur à une approche séquentielle du processus créatif : composer puis jouer. D'un autre côté, les environnements « temps-réels » entremêlent le temps de l'exécution du programme et celui de la restitution des sorties en produisant un signal continu. Ce lien permet une utilisation interactive des programmes, mais limite les capacités de calcul et complique la spécification de structures temporelles.

Cette thèse vise à établir un système informatique permettant le calcul de structures musicales, leurs représentations/manipulations à un niveau compositionnel, ainsi que leurs diffusions interactives. Elle constitue une étude à la croisée de multiples domaines informatiques : la modélisation des systèmes discrets, l'ordonnancement, la conception logicielle ou encore les interfaces homme-machine. Nous proposons une architecture dans laquelle la modification des programmes peut affecter leurs sorties, y compris durant la phase de restitution, tout en conservant les avantages compositionnels de l'approche temps-différé. Nous introduisons également de nouveaux outils pour planifier leur exécution grâce à la conception de scenarios dynamiques, que nous appelons la *meta-composition*. Les mécanismes de calcul et de restitution sont entremêlés : la restitution peut être affectée par des calculs qu'elle a elle-même déclenchés. Les différents résultats décrits dans ce manuscrit sont implantés dans le logiciel de composition OpenMusic, qui peut alors modéliser les œuvres à la fois comme des structures musicales et comme des programmes en continuelle exécution.

ABSTRACT

Computer-aided composition (CAC) tools allow composers to generate musical data using programming. The computer-aided composition programs are generally called *deferred-time* to underline the difference between the time included in their result, and the absolute time that flows during their executions. CAC tools offer a temporal representation of musical data, allowing to compose at the scale of a musical work. However, they do not allow any interaction with this structure during its rendering. This restricts the user to a sequential workflow : compose, then play. On the other hand, *real-time* environments interleave program execution time and output rendering time. This link introduces interactivity in programs, but limits computing power and complexifies the specification of temporal structures.

This thesis aims at designing a computer system enabling the computation of musical structures, their presentation/handling on a compositional side, and their interactive rendering. It is a study at the crossroads between several computer science research fields : discrete systems modeling, scheduling, software design and human-computer interfaces. We propose an architecture where program editing can affect their outputs, including during the rendering phase, while preserving the compositional benefits of the deferred-time approach. Compositions are therefore considered as continually running programs, where computation and rendering mechanisms are interleaved. We introduce new tools and interfaces to arrange their execution through time thanks to dynamic temporal scenario scripting, which we call *meta-composing*. The different results described in this manuscript are implemented in the computer-aided composition environment OpenMusic.

PUBLICATIONS

Journaux internationaux avec comité de lecture

Dimitri Bouche, Jérôme Nika, Alex Chechile et Jean Bresson. **Computer-aided composition of musical processes**, *Journal of New Music Research*, 2016.

Actes de conférences internationales avec comité de lecture

Jérôme Nika, Dimitri Bouche, Jean Bresson, Marc Chemillier et Gérard Assayag. **Guided improvisation as dynamic calls to an offline model**, *Sound and Music Computing conference*, Maynooth, Irlande, 2015.

Dimitri Bouche et Jean Bresson. **Planning and Scheduling Actions in a Computer-Aided Music Composition System**, *9th International Workshop on Scheduling and Planning Applications, International Conference on Automated Planning and Scheduling (ICAPS)*, Jérusalem, Israël, 2015.

Dimitri Bouche, Jean Bresson et Stéphane Letz. **Programmation and Control of Faust Sound Processing in OpenMusic**, *International Computer Music Conference*, Athène, Grèce, 2014.

Actes de conférences nationales avec comité de lecture

Dimitri Bouche et Jean Bresson. **Articulation dynamique de structures temporelles pour l'informatique musicale**, *10ème Colloque sur la Modélisation des Systèmes Réactifs (MSR)*, Nancy, France, 2015.

Jean Bresson, Dimitri Bouche, Jérémie Garcia, Thibault Carpentier, Florent Jacquemard, Diemo Schwarz et John MacCallum. **Projet EFFICACe : Développements et perspectives en composition assistée par ordinateur**, *Journées d'Informatique Musicale*, Montréal, Canada, 2015.

Séminaires invités

Teaching Programming & Music Composition with OpenMusic, *International Conference : "Innovative Tools and Methods to Teach Music and Signal Processing"*, Université Jean Monnet, Saint-Etienne, France, 2015.

Scheduling & Time Structures in Computer-Aided Composition, *CCRMA Guest Colloquium*, Stanford University, USA, 2015.

Scheduling & Time Structures in Computer-Aided Composition, *Latest developments in spatial audio and computer-aided composition tools from Ircam*, University of Oslo, Department of Musicology, Norvège, 2014.

OpenMusic Introduction, *Latest developments in spatial audio and computer-aided composition tools from Ircam*, Norwegian Academy of Music, Norvège, 2014.

Lisp & Music Technology, *European Lisp Symposium*, Ircam, Paris, France, 2014.

OM-Faust : Faust development in OpenMusic, *ANR INEDIT Meeting*, Université de Bordeaux, LaBRI, France, 2013.

Poster

Dimitri Bouche et Jean Bresson. **Adaptive lookahead planning for performing music composition**, *International Conference on Automated Planning and Scheduling (ICAPS)*, Jérusalem, Israël, 2015.

Encadrement

Samuel Bell-Bell : **Étude du contrôle adaptatif de processus dynamiques dans les systèmes multimédia interactifs**, *Stage Master 2 ATIAM*, Ircam/UPMC, France, 2016.

OpenMusic 7 (∼ 10000 lignes de code) — moteur d'exécution (chapitre 3), structures de données (chapitre 4) et interfaces (chapitres 4 et 6) pour la CAO et la metacomposition.

Architecture audio d'OpenMusic (∼ 3000 lignes de code) — articulation dynamique de LibAudioStream (chapitre 5), éditeur audio et table de mixage virtuelle (avec gestion des effets et synthétiseurs Faust).

Bibliothèque OM-Faust (∼ 3500 lignes de code) — compilation et manipulation d'effets et synthétiseurs Faust dans l'environnement OpenMusic (chapitre 5).
(`http://repmus.ircam.fr/openmusic/libraries`)

Improvisation Handler (∼ 1500 lignes de code) *(en collaboration avec Jérôme Nika et al. (2015))* — outils de contrôle interactifs pour l'improvisation guidée avec ImproteK dans OpenMusic.

Bibliothèque OM-ModTile (∼ 600 lignes de code) *(en collaboration avec Hélianthe Caure)* — créer, modifier, visualiser et expérimenter les canons rythmiques compacts modulo 2 dans OpenMusic.
(`http://repmus.ircam.fr/openmusic/libraries`)

REMERCIEMENTS

Cette section sera complétée dans la version finale du manuscrit.

TABLE DES MATIÈRES

INTRODUCTION 1

1 CONTEXTE ET MOTIVATIONS : DE LA CAO À LA META-CAO 5
- 1.1 Composition Assistée par Ordinateur . . . 5
  - 1.1.1 Historique . . . 5
  - 1.1.2 Langages de programmation pour la composition . . . 7
  - 1.1.3 Programmation visuelle . . . 8
- 1.2 OpenMusic . . . 10
  - 1.2.1 Le « patch » : un programme visuel . . . 10
  - 1.2.2 Extension réactive du langage visuel . . . 12
  - 1.2.3 La maquette . . . 13
- 1.3 Systèmes musicaux interactifs . . . 14
  - 1.3.1 Systèmes performatifs (temps-réel) . . . 15
  - 1.3.2 Systèmes interactifs pour la composition : vers une « meta-CAO » 17
- 1.4 Conclusion du chapitre et objectifs . . . 21

2 MODÉLISATION DU TEMPS POUR LA REPRÉSENTATION, L'ORDONNANCEMENT ET LA RESTITUTION DES STRUCTURES MUSICALES 25
- 2.1 Représentation : partition et contenu . . . 25
  - 2.1.1 Temps linéaire, hiérarchique et relations temporelles . . . 26
  - 2.1.2 Précision temporelle d'un système musical . . . 27
- 2.2 Modélisation et écoulement du temps . . . 28
  - 2.2.1 Système musical discret . . . 28
  - 2.2.2 Le temps différé de la composition . . . 28
  - 2.2.3 Le temps réel du jeu . . . 29
  - 2.2.4 Paradigmes temporels synchrone et asynchrone . . . 31
- 2.3 Restitution : mise en temps des structures musicales . . . 33
  - 2.3.1 Planification . . . 33
  - 2.3.2 Ordonnancement . . . 34
  - 2.3.3 Répartition . . . 36
  - 2.3.4 Planification et ordonnancement : application aux systèmes musicaux . . . 36

3 SYSTÈME D'ORDONNANCEMENT POUR LA META-CAO 43
- 3.1 Formalisation des données . . . 43
  - 3.1.1 Actions Musicales . . . 43
  - 3.1.2 Dates . . . 44
  - 3.1.3 Objets musicaux . . . 45

3.1.4 Actions de déclenchements de tâches . . . 47
3.1.5 Trace de restitution . . . 47
3.2 Mécanismes d'exécution . . . 47
3.3 Optimisation pour la meta-composition . . . 48
4 IMPLÉMENTATION : UNE ARCHITECTURE POUR LA REPRÉSENTATION ET LA RESTITUTION DYNAMIQUE DES OBJETS MUSICAUX 55
4.1 Représentation et manipulation des objets musicaux . . . 56
4.1.1 Interface de programmation . . . 56
4.1.2 Éditeur graphique et opérations . . . 56
4.2 Restitution des objets musicaux . . . 57
4.2.1 Méthodes de transport . . . 57
4.2.2 Méthodes d'ordonnancement . . . 59
4.3 Exemples d'implémentation . . . 61
5 GESTION DES RESSOURCES ET DE LA RESTITUTION SONORE 65
5.1 Gestion des ressources audio . . . 65
5.1.1 Généralités . . . 65
5.1.2 Gestion du signal audio dans OpenMusic . . . 66
5.2 Vers une mixité des temps . . . 67
5.2.1 Approche GALS . . . 67
5.2.2 L'exemple OM-Faust . . . 68
5.3 Une architecture audio asynchrone pour les processus compositionnels interactifs . . . 72
5.3.1 Objet son « temps-réel » . . . 72
5.3.2 Implémentation de l'architecture audio . . . 73
5.4 Objets sonores dynamiques . . . 74
5.4.1 Requêtes audio apériodiques : IAE . . . 74
5.4.2 Requêtes audio périodiques : Spat-scene . . . 76
6 FORMALISATION ET IMPLÉMENTATION DES META-PARTITIONS 79
6.1 Sémantique et formalisme . . . 79
6.1.1 Sémantique des patches . . . 79
6.1.2 Sémantique des maquettes . . . 81
6.2 Structure et évaluation d'une meta-partition . . . 81
6.3 Lien au moteur d'ordonnancement . . . 84
6.4 Interface . . . 88
6.5 Programmation et meta-programmation . . . 92
6.5.1 API de la meta-partition . . . 92
6.5.2 Patch de contrôle . . . 93
6.5.3 Évaluations time/event-driven . . . 94
6.6 Exemples . . . 96

6.6.1 Génération musicale dynamique avec spécifications formelles . . 97
6.6.2 Contrôle de la synthèse pour l'oto-émission acoustique 99
CONCLUSION 105

A MODÈLE POUR LA PROGRAMMATION D'OBJETS 107
A.1 Définitions principales 107
A.2 Méthodes de transport 108
B DIAGRAMMES DE SÉQUENCE 109
B.1 s-object « statique » 109
B.2 s-object « dynamique » 110
B.3 Objet sound 111
B.4 Objet iae 112
B.5 Objet spat-scene 114
C PERFORMANCES & TIMING 117
C.1 Validation empirique de l'instantanéité 117
C.2 Optimisation du moteur de calcul 118
C.2.1 Parallélisation et dépendance des tâches 118
C.2.2 Distribution des temps de calcul 120
C.3 Monitoring des retards 122

BIBLIOGRAPHIE 125

INTRODUCTION

Les travaux décrits dans ce manuscrit s'inscrivent dans le champ de l'informatique musicale, discipline axée sur la conception d'outils informatiques pour la création musicale. Ces outils sont divisés en plusieurs catégories où chaque phase de la création musicale est représentée : logiciels dédiés à la notation, à la composition, au concert, au traitement du signal, *etc.* Nous nous intéresserons particulièrement à la composition assistée par ordinateur (CAO), dédiée au calcul de structures musicales.

Les outils de CAO se distinguent des outils liés à la performance musicale. Qualifiés de « temps-différés », ils permettent l'élaboration de processus calculatoires complexes afin de générer des structures musicales en amont de la performance. Ces structures impliquent des représentations et spécifications de contraintes temporelles. Cependant, ces représentations sont indépendantes du temps qui s'écoule lors de l'exécution de ces processus. La qualification temps-différé dénote ainsi une séparation entre le temps de la création et celui de l'exécution des structures musicales (dans la suite de cette thèse, nous parlerons de *restitution*). Les logiciels dédiés à la performance en revanche, sont qualifiés de « temps-réels » et permettent la réalisation de processus génératifs dans le but d'une utilisation interactive (par exemple pendant un concert) au détriment des aspects structurels : le temps des données générées est le même que celui de la performance en cours lors de l'utilisation de ces outils. Ces différentes modélisations du temps sont diamétralement opposées et reflètent une segmentation du processus de production musicale (sous forme de phases successives), ainsi que des divergences de conceptions logicielles.

Cette dualité s'est imposée pour de multiples raisons. Elle est d'abord historique, car les limitations technologiques aux débuts de l'informatique musicale empêchaient l'élaboration d'outils mixtes. Ensuite, parce que les ressources devant être mises à disposition de l'utilisateur sont opposées : les outils de composition doivent allouer de la puissance de calcul quand les outils de performance mettent en avant la précision temporelle. Aujourd'hui, les capacités des ordinateurs grand public ainsi que les pratiques musicales permettent de remettre en cause cette séparation. Les compositeurs ont souvent recours à une utilisation mixte des outils, parfois synchronisée via l'utilisation de protocoles de communication. Ainsi, de multiples extensions des outils temps-réel ont récemment vu le jour afin d'y intégrer des représentations compositionnelles.

Les modèles de calcul de chaque paradigme restent néanmoins difficiles à concilier. Par exemple, le modèle *demand-driven* des outils de CAO – où les calculs sont exécutés à la demande de l'utilisateur – n'est pas directement compatible avec les autres modèles *time*, *data* ou *event-driven* – où les calculs sont déclenchés par le temps, des données

ou des événements. De même, il n'est pas évident d'entremêler des calculs longs à une exécution devant répondre à des contraintes temporelles, comme des échéances ou des allocations de ressources limitées dans le temps.

OBJECTIF. L'objectif de cette thèse est d'élaborer un environnement d'informatique musicale permettant de combiner une représentation des données d'ordre compositionnelle, le calcul et la restitution de structures musicales, et une interaction dynamique avec celles-ci. Les processus de CAO intégrés dans une interaction structurée avec leur contexte sont alors décloisonnés du domaine strictement temps-différé, que ce soit dans une optique compositionnelle ou dans une logique de performance musicale. Bien que des travaux aient été menés en ce sens, il n'existe pas encore de modèle permettant d'unifier ces différents aspects.

Cette problématique englobe donc des questions à la croisée de différents domaines de l'informatique : de l'ordonnancement à la modélisation des systèmes discrets, en passant par les interfaces homme-machine. Elle est au cœur du projet de recherche ANR EFFICAC(e) [1] (Bresson et al., 2015) traitant des rapports entre calcul, temps et interactions en CAO.

Nous proposons un système permettant de fusionner calcul et restitution musicale, dont les principaux composants sont :

- un moteur de calcul et d'ordonnancement ;
- une architecture pour la manipulation et la restitution des structures temporelles ;
- des outils et éditeurs destinés aux compositeurs.

Cette architecture est implémentée comme noyau de calcul et de restitution dans le logiciel OpenMusic. Nous essaierons de démontrer que ce système peut satisfaire différentes utilisations : composition, concert, restitution d'objets musicaux interactifs ou encore affichage dynamique.

1. EFFICAC(e) (*Extended Framework For "In-time" Computer-Aided Composition*) ANR-13-JS02-0004 (2013-2017) est un projet ANR Jeunes Chercheuses et Jeunes Chercheurs (JCJC) proposant une nouvelle orientation dans le développement d'outils de CAO, et plus particulièrement de l'environnement OpenMusic. Une partie de cette recherche a également été effectuée au CCRMA (Center for Computing Research in Music and Acoustics – Stanford University) grâce à une bourse du France-Stanford Center For Interdisciplinary Studies (février-avril 2015).

PLAN DE LA THÈSE. Le chapitre 1 présente notre domaine d'étude qu'est la CAO, et plus particulièrement le logiciel OpenMusic. Nous introduisons les systèmes interactifs musicaux, et expliquons comment CAO et interactivité s'articulent autour du concept de *meta-composition*. Nous définissons finalement cette notion comme la conception de programmes générant des données musicales pouvant être modifiées dynamiquement par des interactions ou par leur propre restitution.

Le chapitre 2 porte sur la modélisation du temps pour la représentation, la manipulation et la restitution des structures musicales. Dans ce chapitre, nous nous focalisons sur les mécanismes temporels des systèmes informatiques. Nous positionnons l'environnement que nous concevons comme une extension d'un langage « hors-temps » vers le temps-réel, et détaillons les mécanismes de planification et d'ordonnancement pouvant s'y intégrer.

Le chapitre 3 présente un système capable d'entremêler calcul, restitution et interaction. Celui-ci est basé sur des modules d'ordonnancement, de calcul et de répartition qui, suivant des comportement réactifs et adaptatifs, assurent une continuité musicale en présence d'interaction avec des processus temps-différés. Au préalable, nous établissons un modèle formel pour la représentation des structures musicales prenant en compte le calcul dans le temps de la restitution.

Nous proposons en chapitre 4 une implémentation permettant la connexion des structures musicales au système présenté en chapitre 3. L'implémentation comprend une interface de programmation et une interface graphique générique pour la manipulation de ces structures.

Le chapitre 5 est consacré à la gestion du signal audio. En effet, dans le cadre de la manipulation de processus temps-différés en temps-réel, les contraintes imposées à la restitution audio nécessitent un traitement particulier. Nous proposons un mécanisme selon lequel un processus de restitution audio indépendant communique de manière asynchrone avec notre système.

Finalement, nous terminons cette thèse au chapitre 6 en donnant une formalisation et les détails d'implémentation d'une *meta-partition* – interface pour la meta-composition – permettant la spécification et l'implémentation de partitions dynamiques interactives. Nous présentons plusieurs exemples musicaux.

I

CONTEXTE ET MOTIVATIONS : DE LA CAO À LA META-CAO

Nous présentons dans ce chapitre le contexte général des travaux décrits dans ce manuscrit : l'informatique musicale, et plus précisément la Composition Assistée par Ordinateur (CAO). La CAO regroupe des outils d'aide à la composition musicale et à la formalisation de processus compositionnels. Cette formalisation passe par l'utilisation de langages de programmation, qui peuvent être des langages informatiques généraux, mais plus souvent des langages conçus pour une utilisation dédiée des paradigmes informatiques, intégrant des structures de données spécialisées pour supporter les représentations musicales. Nous présentons un bref historique de l'informatique musicale en section 1.1 en nous concentrant particulièrement sur la CAO et sur la programmation visuelle. En section 1.2, nous présentons le logiciel de CAO OpenMusic qui sera notre environnement de référence et de développement tout au long de ce travail. La section 1.3 introduit les systèmes musicaux interactifs et la notion de *meta*-CAO. Nous y positionnons la CAO par rapport à la notion d'interactivité, ce qui nous conduit à établir la problématique de cette thèse en section 1.4.

PROCESSUS COMPOSITIONNEL. Dans la suite de ce chapitre et du manuscrit, nous nommerons *processus compositionnel* un programme informatique dont l'exécution produit ou modifie une structure musicale. Nous différencions ce terme du « processus de composition », qui dénote les différentes phases nécessaires à la réalisation d'une œuvre, ou encore d'un « processus musical » qui décrit un phénomène perceptif créé par la restitution de données musicales. La notion de processus compositionnel est étroitement liée à celle de *structure* ou d'*objet musical* : nous considérons que l'objet est le produit d'un processus compositionnel.

1.1 Composition Assistée par Ordinateur

Nous dressons ici une brève chronologie de l'apparition de la CAO. Pour une description plus détaillée, le lecteur pourra se référer à la publication de Rondeleux (1999), au livre de Holmes (2002) ou encore à la chronologie rédigée par Battier et Lemouton.

1.1.1 *Historique*

Le développement de l'informatique au cours du XX ème siècle a rapidement conduit la production des premières musiques par ordinateur. Le CSIR MK1, premier ordinateur

australien, fut également l'un des premiers au monde à générer du son en 1951. Pour cela, Geoff Hill et Trevor Pearcey dirigèrent des impulsions électriques de l'ordinateur vers un amplificateur audio. En faisant transiter des données spécifiques sur le canal dont était extrait le signal, il fut possible de maitriser la hauteur et la durée des sons produits. Cette première expérience musicale numérique ne relève cependant pas encore du domaine de la CAO : ici, la composition est entièrement réalisée par l'homme, l'ordinateur étant uniquement un transcodeur du son. C'est dans les quelques années qui suivirent qu'ont eu lieu les premières expériences en matière de composition assistée numériquement.

> « Music, then, may be defined as an organization of [...] elementary operations and relations between sonic entities or between functions of sonic entities. »
>
> Xenakis (1992)

Comme l'écrit Iannis Xenakis, une composition musicale peut être formalisée comme une organisation d'opérations élémentaires. Cette formalisation peut être réalisée *a posteriori*, permettant ainsi d'analyser une œuvre et de la rendre reproductible, voire paramétrable. Inversement, un compositeur peut formaliser une idée musicale *a priori* de manière à créer un algorithme permettant la génération de données. Cette génération peut être fastidieuse voire impossible à réaliser manuellement quand interviennent un nombre d'opérations ou des espaces combinatoires dont la complexité n'est plus à portée humaine.

L'idée d'implémenter ces processus formalisés en tant que programmes informatiques a émergé dès la fin des années 50. La première composition généralement citée comme exemple d'un tel procédé est l'œuvre *Illiac Suite* (Hiller et Isaacson, 1959) pour quatuor à cordes, réalisée par Lejaren Hiller et Leonard Isaacson en 1956. Ceux-ci ont programmé un ordinateur Illiac I pour le faire générer automatiquement des notes et accords en fonction de règles musicales et de modèles mathématiques.

L'année suivante, le premier logiciel de la série des MUSIC-N faisait son apparition, créé aux Bell Labs par Max Mathews (1963). Les langages de cette série sont dédiés à la synthèse sonore et permettent à un utilisateur de créer des instruments de synthèse (dans divers langages, textuels ou graphiques selon les versions) en interconnectant des modules plus ou moins élémentaires, et de les contrôler via une partition (séquence de notes et de commandes) pour produire un extrait audio-numérique.

1.1.2 *Langages de programmation pour la composition*

> « Nous concevons un tel environnement [de CAO] comme un langage de programmation spécialisé que les compositeurs utiliseront pour construire leur propre univers musical. [...] Ceci nous amène à réfléchir aux différents modèles de programmation existants, aux représentations, internes et externes, des structures musicales qui seront construites et transformées par cette programmation. »
>
> Assayag (1998)

La composition étant, dans le contexte que nous considérons, basée sur un travail de formalisation et d'automatisation de la pensée musicale, les outils de CAO doivent permettre d'implémenter des programmes propres à chaque compositeur. Suivant le modèle des travaux de Mathews, les logiciels de CAO se présenteront donc généralement sous la forme de langages de programmation. Musicomp, créé par Lejaren Hiller (1969), fut l'un des premiers langages dédiés à la composition musicale, implémentant un ensemble de routines facilitant l'expressivité musicale.

Différents types de langages et paradigmes de programmation ont été utilisés pour la CAO. Trois paradigmes majeurs peuvent être cités.

LANGAGES FONCTIONNELS. Le paradigme fonctionnel (Bird et Wadler, 1988) aborde la notion de calcul à travers l'évaluation d'expressions. Il permet de concevoir des programmes comme des compositions de fonctions mathématiques, contrairement à la programmation impérative qui les modélise comme des suites d'instructions et d'affectations. Lisp [1], premier langage qualifié de fonctionnel reposant sur la notion de *lambda-calcul* (Morris, 1969), a été régulièrement utilisé dans la conception d'environnements de CAO. Interprété, il permet une utilisation interactive. Sa représentation des données sous forme de listes chaînées accompagnée d'un ramasse-miette – qui permet de s'abstraire de la gestion mémoire – en fait également un support idéal pour l'informatique musicale (manipulation aisée de séquences mélodiques, rythmiques, *etc.*). Certains langages fonctionnels, comme Haskore (Hudak et al., 1995) – basé sur le langage Haskell [2] – mettent en avant une approche algébrique, qui a pu ainsi également s'appliquer au domaine musical.

LANGAGES ORIENTÉS OBJETS. Le paradigme objet permet d'organiser les données dans des structures modélisant des concepts, et de définir des relations entre eux (Rumbaugh et al., 1991). Les notions d'objet et d'héritage facilitent la représentation de structures musicales complexes. Le polymorphisme, le typage et les redéfinitions permettent au compositeur de modéliser des problèmes et un univers musical de manière

1. http://www-formal.stanford.edu/jmc/history/lisp/lisp.html
2. https://wiki.haskell.org/Introduction

structurée et intuitive. Nous pouvons citer ici les travaux précurseurs de l'environnement Dmix d'Oppenheim (1989).

PROGRAMMATION PAR CONTRAINTES. Les langages par contraintes (Rossi et al., 2006) ont également prouvé leur utilité pour exprimer des problèmes musicaux (Truchet et Assayag, 2011). De nombreuses disciplines de la théorie musicale peuvent être modélisées comme un ensemble de contraintes : le contrepoint, l'harmonie, l'instrumentation *etc.* Différents systèmes de programmation musicale par contraintes ont été développés (Anders et Miranda, 2011), offrant des modules reprenant ces concepts de base et/ou permettant d'exprimer des problèmes d'optimisation combinatoire divers liés aux structures musicales.

Chaque paradigme possédant ses atouts spécifiques, et chaque problème se prêtant plus ou moins bien à une résolution dans chacun d'eux, certains environnements proposent une association de paradigmes, comme par exemple SuperCollider (Wilson et al., 2011) ou encore l'environnement OpenMusic (voir section 1.2) qui supportent les trois paradigmes cités précédemment (Agon, 2004).

1.1.3 *Programmation visuelle*

Un langage de programmation visuelle est un langage offrant une interface à plus d'une dimension pour la programmation. La sémantique d'un programme dans un langage n'est donc plus uniquement déterminée par du texte, mais par des agencements graphiques.

Les environnements de programmation visuelle dédiés à la CAO sont majoritairement basés sur la construction de graphes et permettent de construire des programmes par agencement et connexion de composants fonctionnels dans un espace bi-dimensionnel. C'est le cas de PatchWork (Laurson et Duthen, 1989, voir figure 1.1) et ses descendants OpenMusic (Assayag et al., 1999) et PWGL (Laurson et al., 2009). [3] Ces trois environnements sont basés sur le langage Lisp. Il permettent de construire des programmes visuels correspondant à des *s-expressions* [4] (voir figure 1.2) dont les données peuvent être éditées interactivement.

D'autres représentations graphiques sont possibles, comme le montre par exemple Elody (Letz et al., 1997), environnement de programmation visuelle dérivé du lambda calcul. Dans cet environnement, l'utilisateur peut créer des objets musicaux en asso-

3. C'est aussi le cas de Max (Puckette, 1991) et PureData (Puckette, 1996), deux environnements de programmation visuelle dédiés aux applications musicales temps-réel, sur lesquels nous reviendrons par la suite.

4. Expressions exprimées sous forme de listes imbriquées.

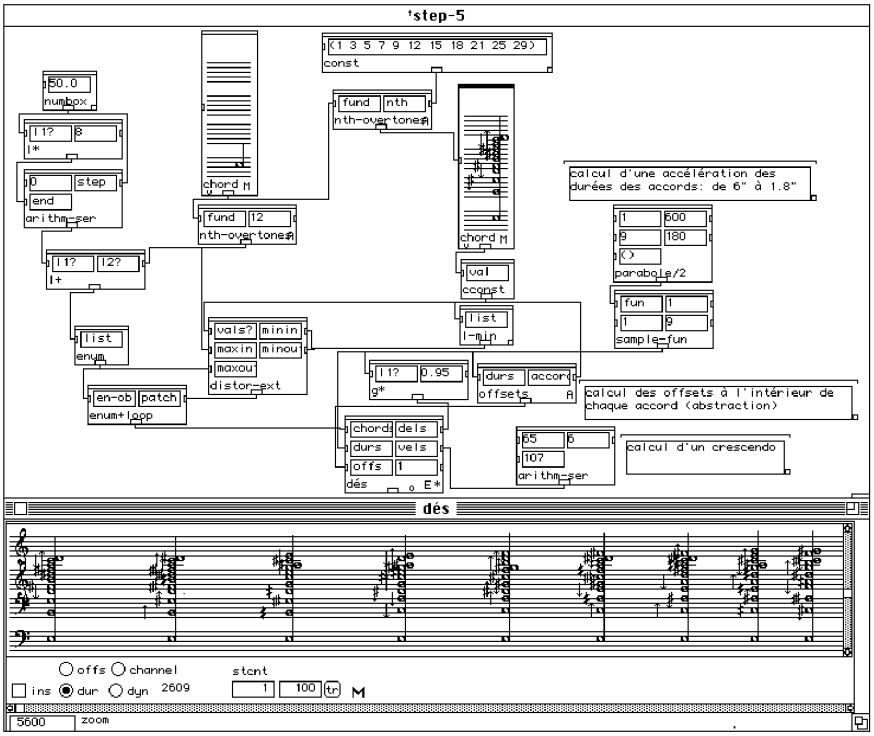


Figure 1.1 – Exemple de programme visuel (patch) dans le logiciel PatchWork (1989).

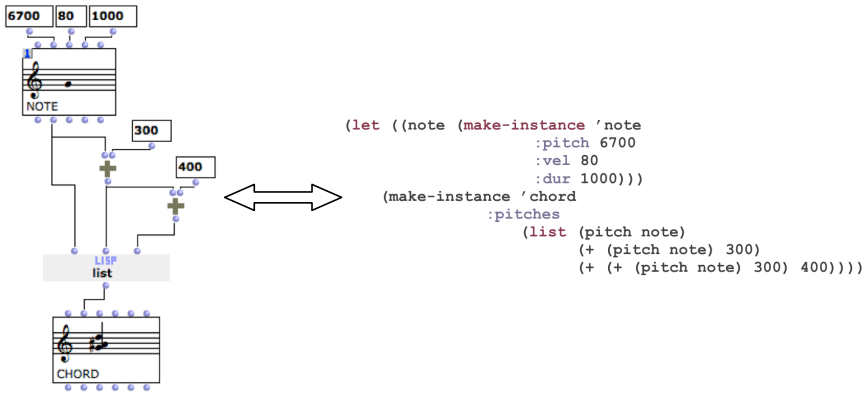
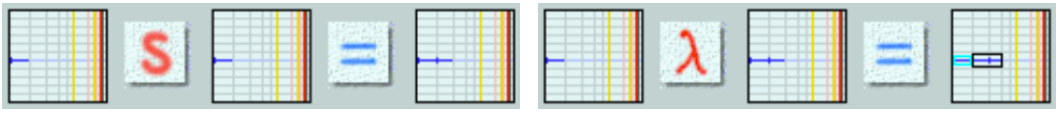


Figure 1.2 – Équivalence entre un programme visuel et une expression Lisp dans OpenMusic.



(a) Mise en séquence de deux notes (b) Création d'un opérateur

Figure 1.3 – Exemples de programmation visuelle dans Elody, figures extraites de Letz et al. (1997).

ciant des données élémentaires (par exemple une séquence de deux notes) à des opérateurs (par exemple « répétition ») – voir figure 1.3a. Ensuite, utilisant le paradigme du *lambda-calcul*, celui-ci peut créer ses propres opérateurs en abstrayant des paramètres dans une expression – voir figure 1.3b.

1.2 OpenMusic

> OpenMusic (OM) est un langage de programmation visuelle basé sur le langage Common Lisp, héritier de PatchWork et développé par l'équipe Représentations Musicales à l'Ircam. Cet environnement fournit une interface graphique permettant l'utilisation visuelle des principales constructions du langage Lisp (abstractions, fonctions d'ordre supérieur, récursivité *etc.* – Bresson et al., 2009), et enrichit celui-ci avec des objets musicaux et des opérateurs spécifiques. Un programme visuel dans OpenMusic représente une expression fonctionnelle que le compositeur peut *évaluer* (calculer) pour générer des structures musicales (ou d'autres types de données). Il correspond à notre définition précédente du processus compositionnel.

1.2.1 *Le « patch » : un programme visuel*

La programmation visuelle dans OpenMusic s'opère par la manipulation des fonctions et données sous forme de *boîtes*, ainsi que par l'utilisation d'interfaces graphiques pour l'édition de valeurs.

Les *boîtes fonctionnelles* font référence à des fonctions, et possèdent un certain nombre d'entrées et de sorties correspondant respectivement aux paramètres et aux sorties de ces fonctions (une fonction peut être multivaluée). Les *boîtes d'objets* font quant à elle référence à une classe d'objet, leurs entrées aux valeurs d'initialisation des attributs de la classe (pour le constructeur) et leurs sorties aux accesseurs de ces mêmes attributs. Les *boîtes de valeurs* sont utilisées pour la manipulation visuelle de valeurs constantes de tout type (par exemple des entiers, réels, chaînes de caractères, *etc.*). Enfin, les *boîtes d'abstractions* sont semblables aux boîtes fonctionnelles, mais font référence à un autre programme visuel, permettant à l'utilisateur de développer

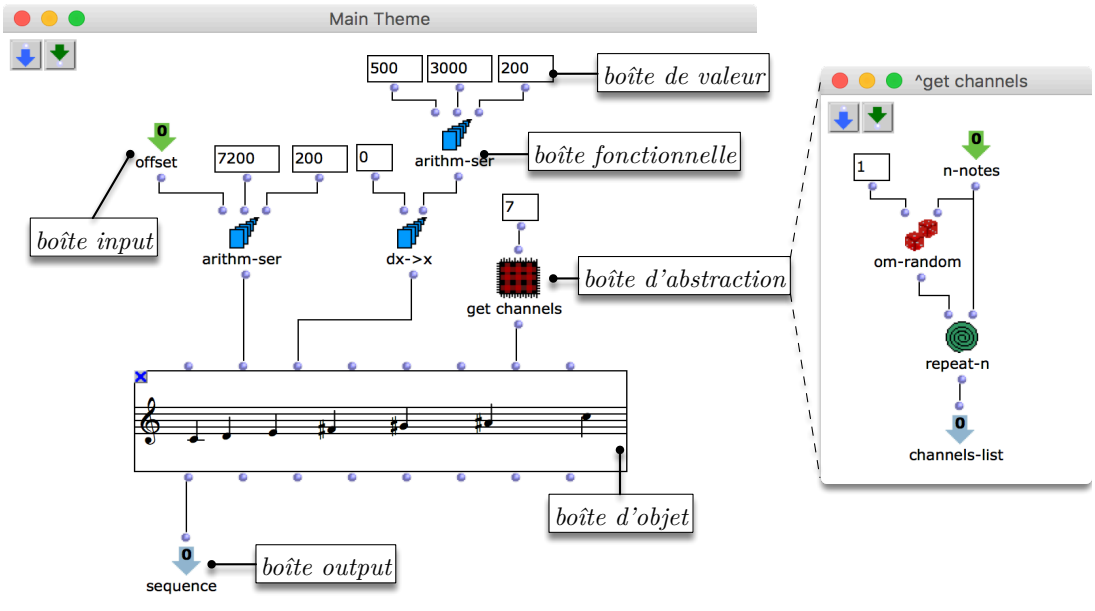


Figure 1.4 – Exemple de *patch* dans OpenMusic.

visuellement des fonctions qu'il pourra utiliser dans d'autres programmes. Nous désignerons par la suite ces boîtes d'abstractions directement par le terme *abstractions*.

Les boîtes peuvent être connectées les unes aux autres, constituant un graphe orienté acyclique nommé *patch*. Une connexion entre une sortie d'une boîte a et une entrée d'une boîte b dans ce graphe implique qu'un argument de b requiert une valeur produite par a. Ainsi, les connexions entre les boîtes représentent la composition des fonctions et le flux des données dans un processus.

ABSTRACTION. Pour être utilisé en tant qu'abstraction, un patch doit présenter un certain nombre de points d'entrées et de sorties, modélisés par les boîtes spéciales *input* et *output* (ne faisant référence à aucune fonction), que l'utilisateur peut intégrer en nombre arbitraire. Une abstraction permet d'utiliser un patch comme une fonction : l'utilisation des boîtes *input* permet de spécifier ses arguments, et les boîtes *output* de récupérer ses résultats. Par exemple, le patch présenté en figure 1.4 est équivalent à une fonction à un argument (nommé *offset*) et un résultat (nommé *sequence*).

La fonction *get-channels* en figure 1.4 est un exemple de patch utilisé en tant que fonction dans un autre contexte sous forme d'une abstraction. Le travail de composition peut ainsi être modularisé et constitué d'un nombre arbitraire d'imbrications fonctionnelles.

La création de patches par les compositeurs dans OpenMusic est souvent expérimentale : il est commun que les boîtes *input* et *output* ne soient pas utilisées, ou encore

que le graphe d'un patch ne soit pas connexe, c'est à dire qu'il existe plusieurs sous-graphes utiles à l'intérieur de celui-ci. Cependant, nous ne nous attarderons pas sur cette utilisation : nous considérerons les patches comme des programmes génératifs intégrés au sein d'une partition, ce qui revient à supposer que tout patch possède un nombre arbitraire d'entrées et au moins une sortie. Le patch en figure 1.4 correspond par exemple à cette définition, décrivant une fonction produisant un résultat à partir d'un argument.

EVALUATION. L'évaluation d'un patch consiste à calculer une valeur pour chacune de ses boîtes. Une évaluation dans OpenMusic est déclenchée explicitement par l'utilisateur. Le mécanisme est récursif et se propage d'une boîte source (par exemple, la boîte de sortie) en direction des boîtes situées en amont dans le graphe. Le flot de contrôle décrit par un programme visuel est donc ascendant et qualifié de *demand-driven.* [5]

Dans sa version la plus simple, un patch est entièrement recalculé lors d'une évaluation. Cependant, il n'est pas toujours nécessaire d'exécuter toutes les opérations qu'il décrit : certains résultats obtenus lors d'évaluations précédentes peuvent être conservés. Pour plus de détails sur les mécanismes d'évaluation dans OpenMusic, le lecteur pourra se référer à Bresson et Giavitto (2014).

1.2.2 ..... *Extension réactive du langage visuel*

Les travaux de Bresson et Giavitto (2014) ont permis d'étendre le paradigme de programmation visuelle dans OpenMusic afin d'intégrer des fonctionnalités réactives, étendant le mode d'évaluation « à la demande » des programmes visuels par un mode d'exécution guidée par des événements (par exemple la modification d'un de ses paramètres, ou encore la réception d'un message envoyé par une autre application). La figure 1.5 illustre ces deux modes d'exécution d'un processus compositionnel.

PROCESSUS COMPOSITIONNEL RÉACTIF. Nous définissons un « processus compositionnel réactif » comme un processus compositionnel au sein duquel la modification d'un paramètre entraîne une réaction qui déclenche son évaluation (et donc modifie son résultat).

5. L'évaluation d'un programme visuel dans OpenMusic est en réalité équivalent à celle de l'expression fonctionnelle équivalente en Lisp.

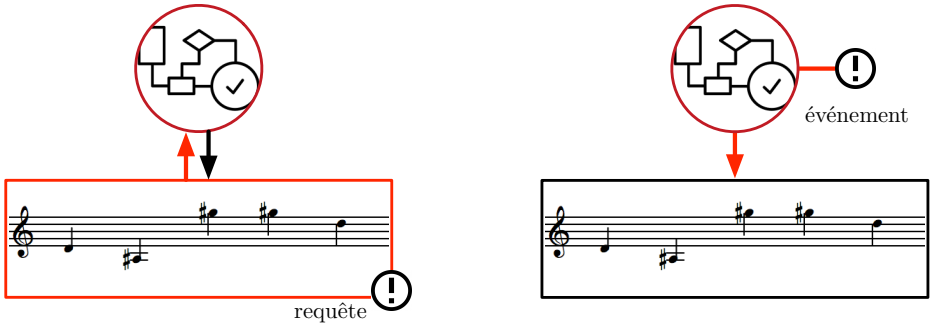


Figure 1.5 – Évaluation d'un processus compositionnel à la demande (à gauche) et réactif (à droite).

Par la suite, nous distinguerons les évaluations selon les termes suivants :

- *demand-driven* : évaluation déclenchée par une demande, par exemple une requête de l'utilisateur.
- *event-driven* : évaluation déclenchée par un événement, par exemple la réception d'un nouveau paramètre.
- *time-driven* : évaluation déclenchée par le temps, par exemple périodique ou lorsqu'une date est atteinte par une horloge.

1.2.3 *La maquette*

Dans OpenMusic, une *maquette* (voir figure 1.6) est une extension du patch dans laquelle la dimension horizontale représente le temps. La position des boîtes dans les deux dimensions graphiques (verticale et horizontale) peut être utilisée comme paramètre et affecter leurs valeurs. La maquette est donc un environnement hybride, unifiant les notions de programme visuel et de partition, et rassemblant ainsi deux formes d'exécutions d'un processus musical : son calcul (ou « évaluation ») et sa restitution, qui désigne l'exécution dans le temps de la structure musicale (production de sons ou d'actions de contrôle).

L'évaluation de la maquette est nécessairement antérieure à sa restitution : elle évalue tous ses sous-programmes et collecte leurs résultats, généralement des objets musicaux, dans une structure de plus haut niveau pouvant être restituée (sous forme sonore) ou utilisée par d'autres programmes. Nous verrons en section 2.3.4.1 que ce mécanisme correspond à une stratégie d'« ordonnancement statique ». Nous verrons aussi qu'en utilisant des mécanismes réactifs, la maquette nous permettra de développer des systèmes musicaux interactifs.

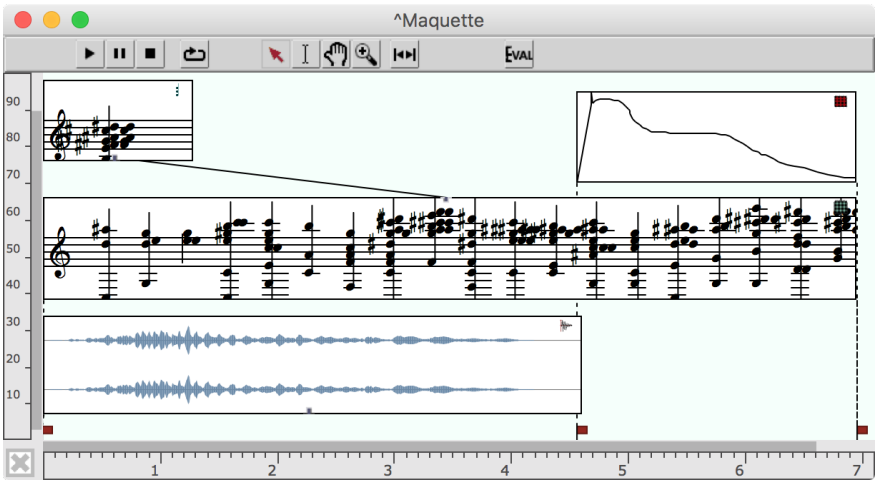


Figure 1.6 – Exemple de maquette dans OpenMusic.

1.3 Systèmes musicaux interactifs

L'interaction entre deux entités désigne une influence mutuelle entre celles-ci, induisant des changements dans leurs comportements. Chadabe (1977) caractérise les « systèmes musicaux interactifs » comme des systèmes à rétroaction où la sortie audio influence l'utilisateur (compositeur, interprète ...) par ce qu'il perçoit pendant qu'il le contrôle (voir figure 1.7).

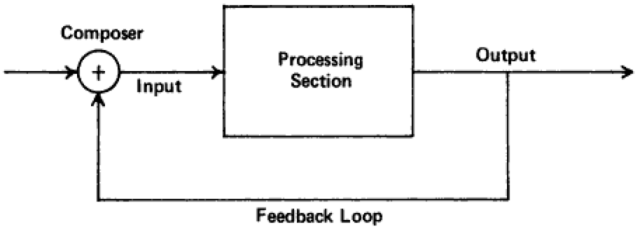


Figure 1.7 – Modèle de système musical interactif, par Chadabe (1977).

Ces systèmes se retrouvent par exemple dans la musique mixte (Tiffon, 1994), où la partie électronique est guidée par le jeu du/des instrumentiste/s, et dont le son produit influence en retour ce jeu. Nous pouvons également citer les orchestres d'ordinateurs portables (Wang et al., 2009), ou encore les installations sonores (Coulter-Smith, 2006).

L'interactivité permet de connecter les domaines de la composition et de la performance. Par exemple, des aspects compositionnels peuvent intervenir dans la perfor-

mance par le biais d'esquisses enregistrées en temps-réel et injectées par la suite dans l'œuvre (cf l'utilisation de la technique de *looper*[6] dans la musique populaire).

1.3.1 *Systèmes performatifs (temps-réel)*

En général, les systèmes interactifs musicaux sont dédiés à être utilisés en situation de concert (on pourra parler de « perfomance » et de « systèmes performatifs »). Le développement de tels systèmes à l'Ircam a été porté par la conception de la première station audio-numérique, la 4X, par Giuseppe Di Giugno. Celle-ci permettra au logiciel Patcher (Puckette, 1988), contrôlant la 4X via le protocole MIDI, de voir le jour. Ce logiciel est l'ancêtre de Max (Puckette, 1991), environnement le plus utilisé actuellement en musique contemporaine. Max (ainsi que sa déclinaison *open-source* PureData – Puckette, 1996) est basé sur le paradigme de *patching* introduit précédemment avec les environnements de cao (voir figure 1.8) fournissant à l'utilisateur un ensemble de modules associant outils mathématiques et traitement du signal en temps-réel.

Contrairement à OpenMusic, ces environnements reposent sur un paradigme *dataflow*, où toute modification d'une donnée se propage automatiquement dans le programme et en modifie les valeurs. Ils opèrent en parallèle à une couche audio synchrone, permettant ainsi de réagir à l'occurence d'événements et de contrôler en conséquence des processus de synthèse ou de traitement du signal. On parle donc d'exécution à la fois *time-driven* – car dirigée par l'écoulement du temps (partie audio) – et *event-driven* – car réactive aux événements (partie contrôle). Les programmes s'exécutent en continu[7] : ce modèle est ainsi particulièrement adapté au concert, car il permet une gestion temps-réel (voir section 2.2.3) des signaux entrants (interactions avec l'environnement et l'utilisateur) et sortants (signaux audio synthétisés).

Moyennant l'implantation de structures de données adéquates, les environnements *dataflow* temps-réel comme Max permettent la réalisation de processus compositionnels réactifs tels que nous les avons définis précédemment (voir section 1.2.2). C'est le cas notamment de la bibliothèque *bach: automated composer's helper* (Agostini et Ghisi, 2013), qui intègre des notations et modules de traitement de données musicales symboliques au sein de l'environnement temps-réel Max. Ainsi, l'utilisateur peut observer des structures de « long terme » tout en modifiant les paramètres des calculs qui les produisent. La figure 1.9 montre un patch réalisé avec cette bibliothèque.

6. Outils électroniques permettant l'enregistrement de matériel sonore et sa répétition automatique en boucle, afin d'opérer des superpositions.

7. En réalité, l'exécution opère pas à pas, guidée par une génération du signal audio par *buffer*, comme nous le verrons en chapitre 5.

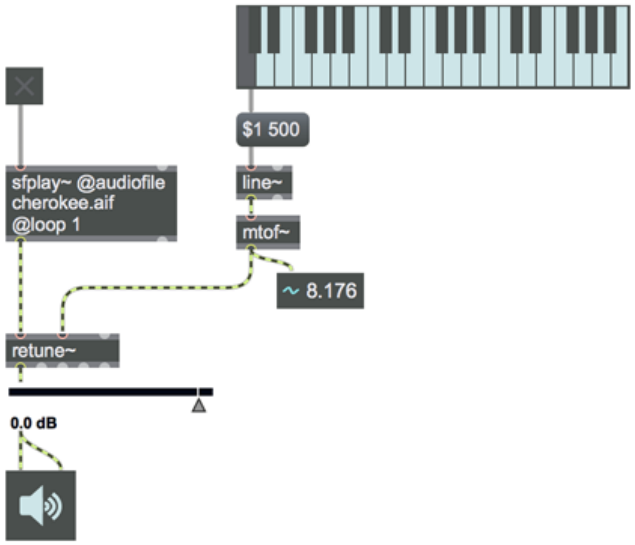


Figure 1.8 – Exemple de patch dans le logiciel Max pour altérer la hauteur d'un fichier audio en temps-réel.

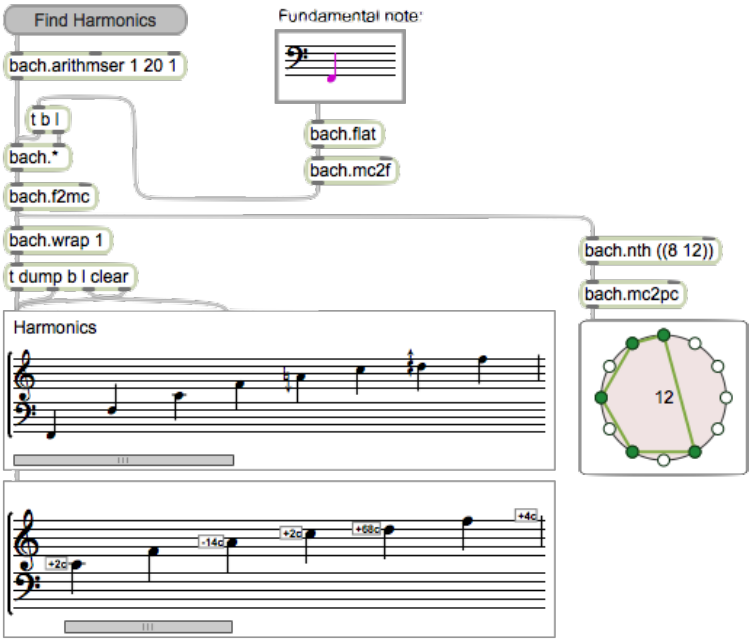


Figure 1.9 – Exemple de «processus compositionnel réactif » réalisé avec la bibliothèque Max *bach: automated composer's helper* pour calculer les harmoniques d'une note.

1.3.2 ..... *Systèmes interactifs pour la composition : vers une « meta-*CAO *»*

L'intégration de capacités interactives dans les systèmes dédiés à la composition pose des problèmes spécifiques. Les travaux de Joel Chadabe (1984) sont pionniers dans le domaine. Utilisant le CEMS (*Coordinated Electronic Music Studio*, un système musical analogique programmable construit par Robert Moog), il divise le processus de composition en deux étapes : d'abord, le compositeur établit des règles de génération musicale (un programme) ; ensuite, il interagit avec ce même programme en utilisant des entrées analogiques. Le programme agissant sur une structure musicale (et non seulement sur une sortie sonore instantanée), les actions de l'utilisateur entraînent des réactions affectant le long terme de la génération. Les conséquences des actions n'étant pas totalement prédictibles, la sortie du système peut influencer l'utilisateur. Nous pouvons voir ici que la notion de « composition interactive » exprime le « contrôle temps-réel de processus compositionnels ».

Comme le montre cet exemple, la composition en temps-réel ne se résume pas au travail à la hâte d'un compositeur pendant une performance. L'aspect structurel et la portée temporelle d'une composition nécessitent un travail de formalisation antérieur à la situation de concert, y compris pour les structures calculées dynamiquement. C'est l'idée de *meta-composition* que nous développons ci-dessous.

> « En reléguant des responsabilités créatives à l'interprète et au programme informatique, le compositeur pousse la composition à un niveau supérieur (à un meta-niveau décrit dans les processus exécutés par le compositeur) et l'externalise (via l'instrumentiste improvisant suivant les règles établies). »
>
> Traduit de Rowe (1996)

META-COMPOSITON. La *meta-composition* est un travail où le compositeur conçoit un programme chargé de prendre en compte des interactions lors de la performance afin de générer du matériel musical dynamiquement. La différence avec la composition, dans les environnements de CAO par exemple, est que les données musicales sont vouées à être modifiées pendant leur restitution, par ces mêmes règles qui en ont permis leur calcul.

Une illustration du processus de meta-composition face au processus de composition classique est disponible en figure 1.10. Dans la branche supérieure, la partition est vue comme des données brutes (résultat d'un processus) ; dans la branche inférieure, elle est vue comme un programme (ensemble de données, processus et règles) en cours d'exécution.

La notion de meta-composition est souvent utilisée pour parler d'improvisation musicale. Nika et al. (2016) l'utilisent pour qualifier le système d'improvisation automatique guidée ImproteK. Une œuvre créée grâce à ce système peut être qualifiée de

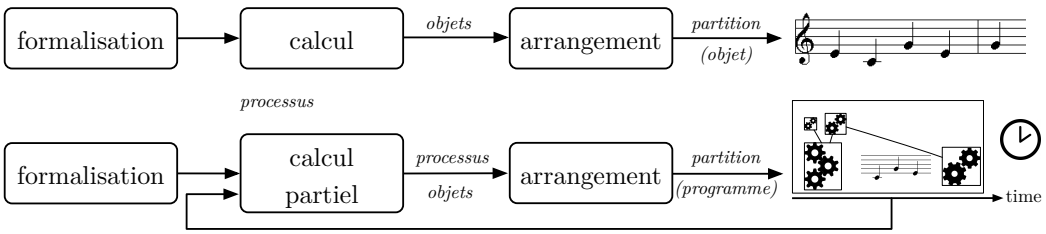


Figure 1.10 – Processus de composition (en haut) et processus de meta-composition (en bas).

meta-composition car ImproteK modélise l'improvisation comme une recombinaison d'improvisations passées, suivant un scénario arbitraire (progression harmonique) dont l'alphabet peut être spécifié par l'utilisateur. Ainsi, un compositeur peut préparer une œuvre en lui donnant un ensemble de règles à respecter, et laisser le système la créer ensuite en temps-réel en interaction avec un musicien.

Une idée proche de cette notion de meta-composition fut mise en œuvre par exemple dans le système d'informatique musicale Formes (Rodet et Cointe, 1984), langage conçu afin de manipuler des processus (*objets actifs* décrits par Cointe, 1983) pour le contrôle de synthèse. Un concept similaire a été mis en avant plus récemment par l'environnement de composition musicale Elody (introduit en section 1.1.3), où des processus traitant un signal entrant en temps-réel peuvent être manipulés sous la forme d'objets statiques sur une frise chronologique.

Dans une certaine mesure également, cette spécificité est présente dans des projets comme la bibliothèque *bach* mentionnée précédemment. Le package *cage* (Agostini et al., 2016) introduit notamment la notion de meta-information dans les objets musicaux de la partition : une note peut par exemple déclencher un processus « caché » supplémentaire lors de son exécution, comme une courbe d'automation.

META-PARTITION. Nous nous intéresserons enfin à la notion de partition en meta-composition, faisant echo à des travaux précédents sur les « partitions interactives » ou « dynamiques ». Différents environnements musicaux font explicitement référence à ces concepts :

- i-score (Baltazar et al., 2014) est un séquenceur interactif pour la création intermédia. Ce logiciel permet à un utilisateur de définir des scénarios temporels d'événements et automations, avec des contraintes et relations chronologiques entre les objets. Le séquenceur est construit sous une forme « multi-linéaire » : chaque objet possède sa propre temporalité. Bien que permettant de définir des relations temporelles, i-score ne propose cependant pas d'outils génératifs pour la création des structures de contrôle.
- Antescofo (Cont, 2008) est un système de suivi de partition couplé à un langage de programmation synchrone pour la composition musicale (Echeveste et al., 2013). Il est

utilisé pour synchroniser une performance instrumentale avec des actions électroniques. L'utilisateur peut programmer des processus avec des temporalités distinctes, dépendantes de n'importe quelle variable à laquelle il a accès (comme l'estimation de tempo du musicien par exemple). Cela revient à construire une partition électronique augmentant la partition de l'instrumentiste, et permettant de créer des relations temporelles fines entre les deux. La partition créée peut être qualifiée d'interactive car l'écoulement du temps auquel elle est soumise est dépendant des informations temporelles de l'instrumentiste (sa vitesse de jeu notamment). Nous y reviendrons au chapitre 2.

- PureData (voir section 1.3.1) propose un pont entre les capacités temps-réel de Max, et une représentation des données d'ordre compositionnelle. Pour cela, l'environnement fournit des outils de programmation et de visualisation de structures de données (Puckette, 2002). Les positions de ces structures dans un espace graphique peuvent être liées au temps et permettre l'élaboration de partitions graphiques (voir figure 1.11). L'utilisateur peut ensuite interagir en temps réel avec les objets graphiques, donc sur les structures qu'ils représentent et sur la sortie audio du système.

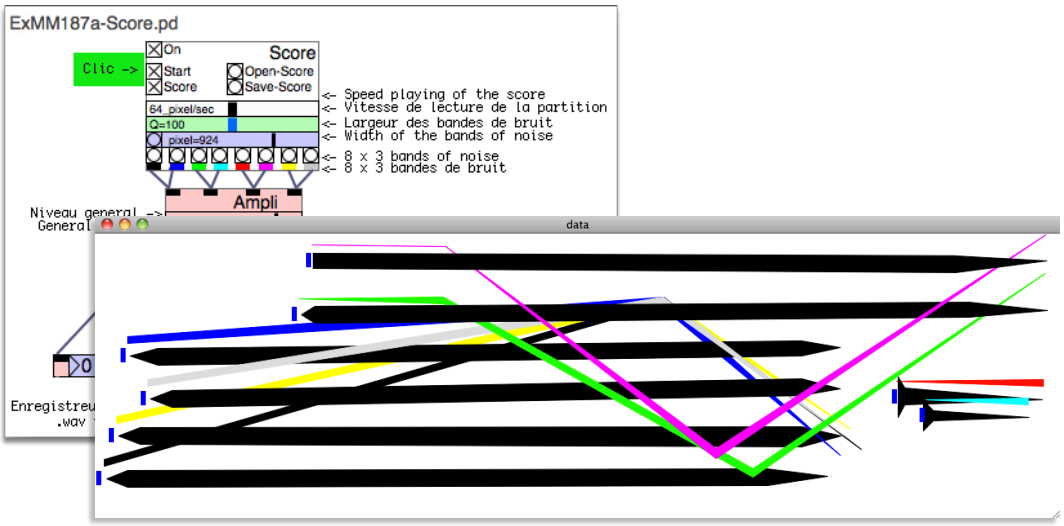


Figure 1.11 – Partition réalisée dans PureData par Gérard Paresys (`http://gerard.paresys.free.fr/Theme/Theme8.html`).

Dans la suite de ce manuscrit, nous qualifierons de *meta*-CAO les outils informatiques destinés à faciliter la réalisation de meta-compositions. Pour illustrer plus clairement ce terme, nous donnons un exemple schématique reprenant les concepts que nous venons d'évoquer.

exemple. Les figures 1.12 à 1.15 présentent la restitution d'une réalisation de meta-composition. Dans la partition présentée cohabitent trois pistes qui seront restituées par un système de meta-cao :

- les deux premières pistes contiennent des données musicales modifiées dynamiquement : des notes MIDI, un fichier audio et un objet se remplissant progressivement de messages de contrôles, afin de modifier les paramètres d'un effet altérant la sortie audio ;
- la troisième piste contient un accompagnement statique : des notes MIDI.

Les éléments étiquetés en rouge sont produits par le compositeur (un objet O et quatre processus compositionnels P1… P4) dans l'environnement de meta-cao. Ceux étiquetés en bleu sont les résultats d'exécutions de certains de ces processus. La restitution de la partition est destinée à boucler un nombre arbitraire de fois.

• La figure 1.12 montre l'état initial de la partition. P1 et P4 ne sont pas évalués avant le démarrage de la restitution, mais placés sur la partition afin de générer du contenu dynamiquement. Seuls P2 et P3 ont produit au préalable les objets O2 (fichier audio) et O3 (initialement vide).

• En figure 1.13, la restitution de la partition est en cours et atteint la date à laquelle le processus compositionnel P1 doit être évalué pour générer une phrase musicale O1. L'exécution de ce programme dépend de règles définies par le compositeur et du contenu de l'objet O2 (échantillons audio).

• En figure 1.14, le programme P2 reçoit un événement utilisateur qui entraîne la synthèse d'un nouveau fichier venant remplacer l'objet O2. Son calcul prend un certain temps, mais n'a pas d'incidence sur l'écoulement de la restitution.

• En figure 1.15, la date à laquelle P4 doit être évalué est atteinte. Son exécution déclenche le calcul de P1 (situé dans le passé de la restitution) qui produit alors une nouvelle séquence de notes, O2 ayant été modifié à l'étape précédente.

• A chaque étape, on peut remarquer que l'objet O3 est « alimenté » par P3 : ce processus s'exécute périodiquement afin de construire des messages de contrôle à partir des valeurs reçues par un capteur physique.

Remarquons que les modifications des données de la partition peuvent à la fois être antérieures ou postérieures au temps de la restitution : une partition est toujours considérée dans son entière temporalité, y compris pendant sa restitution. L'une des autres caractéristiques importantes de la meta-composition que nous souhaitons illustrer ici est la multiplicité des mécanismes de calcul :

- *demand-driven* en figure 1.12, car l'utilisateur a demandé explicitement à P2 de fournir un résultat avant le démarrage de la restitution ;
- *time-driven* en figures 1.13 et 1.15, où les calculs de P1 et P4 sont déclenchés par l'écoulement du temps ;

– *event-driven* en figure 1.14, où un événement déclenche le calcul de P2. P3 est évalué initialement sous forme d'un objet vide mais est alimenté au long de la restitution suivant le modèle *event-driven*.

1.4 Conclusion du chapitre et objectifs

Nous qualifions l'exemple précédent de meta-composition car le compositeur compose non seulement du matériel musical, mais aussi des règles et mécanismes qui permettront de (re)générer des données en temps-réel, bien que les évaluations des processus compositionnels soient temps-différés (de durées arbitraires et non nécessairement interruptibles). Ce procédé se distingue de la composition générative ou algorithmique car une partition peut être re-générée pendant sa restitution : elle possède un ensemble d'états arbitraires.

Nous envisageons donc une partition comme une structure dont le contenu n'est jamais « figé », mais qui consiste plutôt en un programme dont la sortie peut évoluer dynamiquement au cours du temps, pendant sa restitution, suivant les actions d'un utilisateur, des événements extérieurs, et des règles prédéfinies. Une telle partition est donc à la fois ***interactive*** (pouvant être contrôlée), ***dynamique*** (pouvant évoluer automatiquement à la suite d'interactions avec l'environnement) et ***observable*** (pouvant être visualisée) sur la totalité de sa portée temporelle.

Le but fondamental de cette étude est d'essayer de combler le fossé existant entre les environnements dédiés à la composition et ceux dédiés à la performance (Puckette, 2004). Nous souhaitons porter l'idée de meta-composition en créant un environnement et des outils permettant de simplifier le contrôle de processus musicaux en temps-réel, mais aussi d'organiser et de connecter des processus compositionnels afin de construire des structures musicales qui évoluent dynamiquement, suivant des règles automatiquement traitées ou des interactions avec un ou plusieurs utilisateurs.

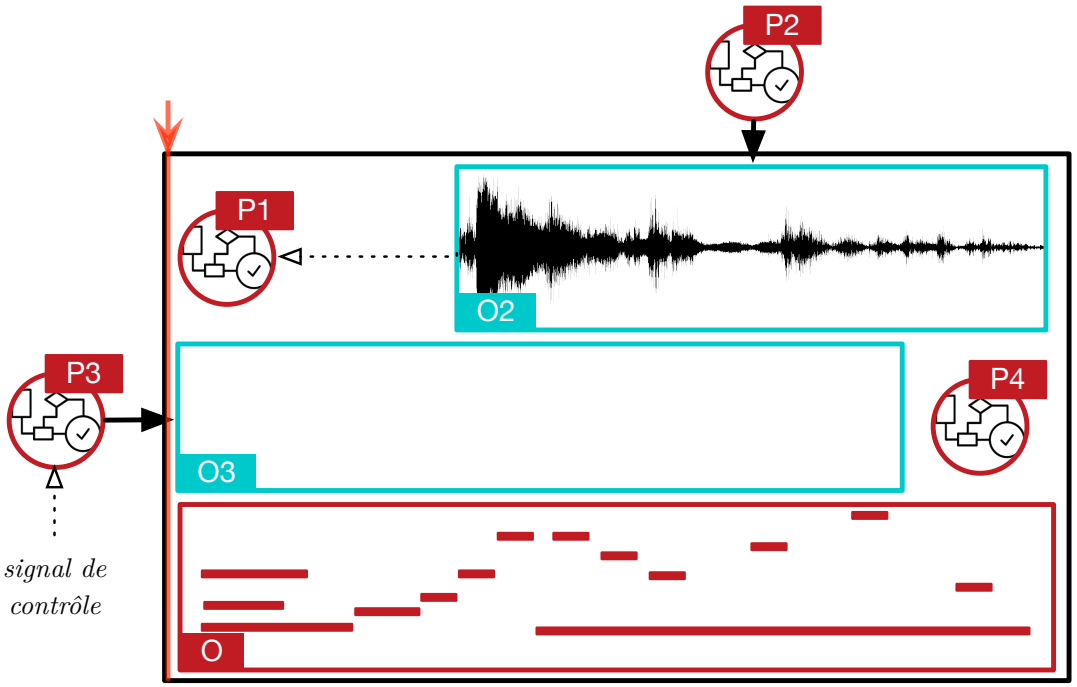


Figure 1.12 – Exemple de meta-composition : état de la partition pré-restitution.

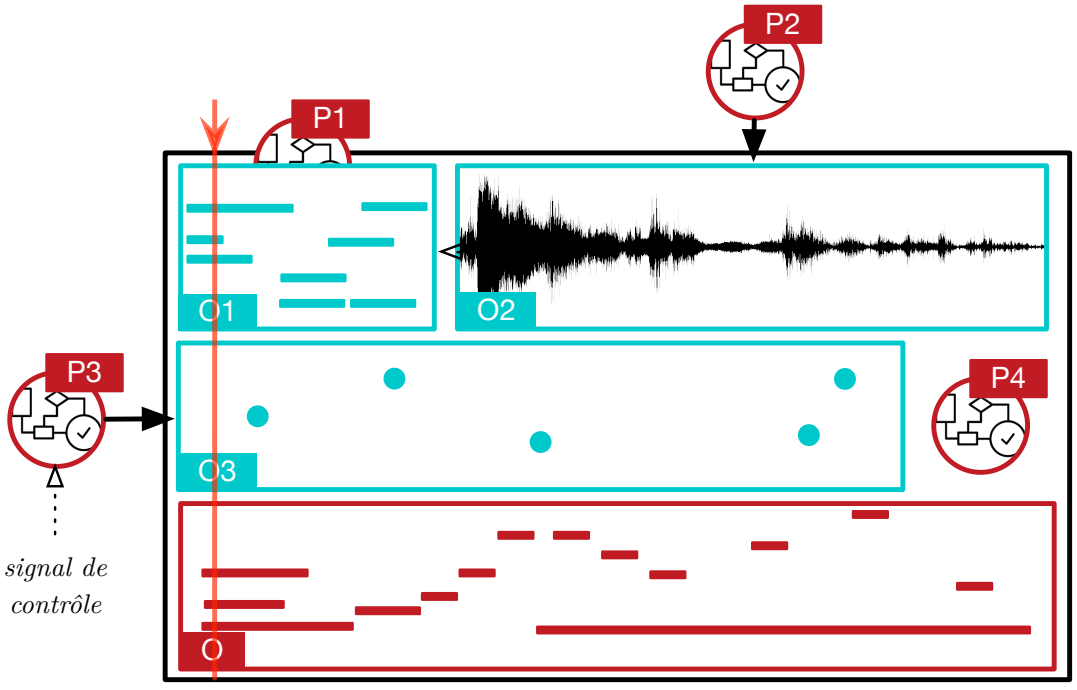


Figure 1.13 – Exemple de meta-composition : état de la partition après l'évaluation de P1.

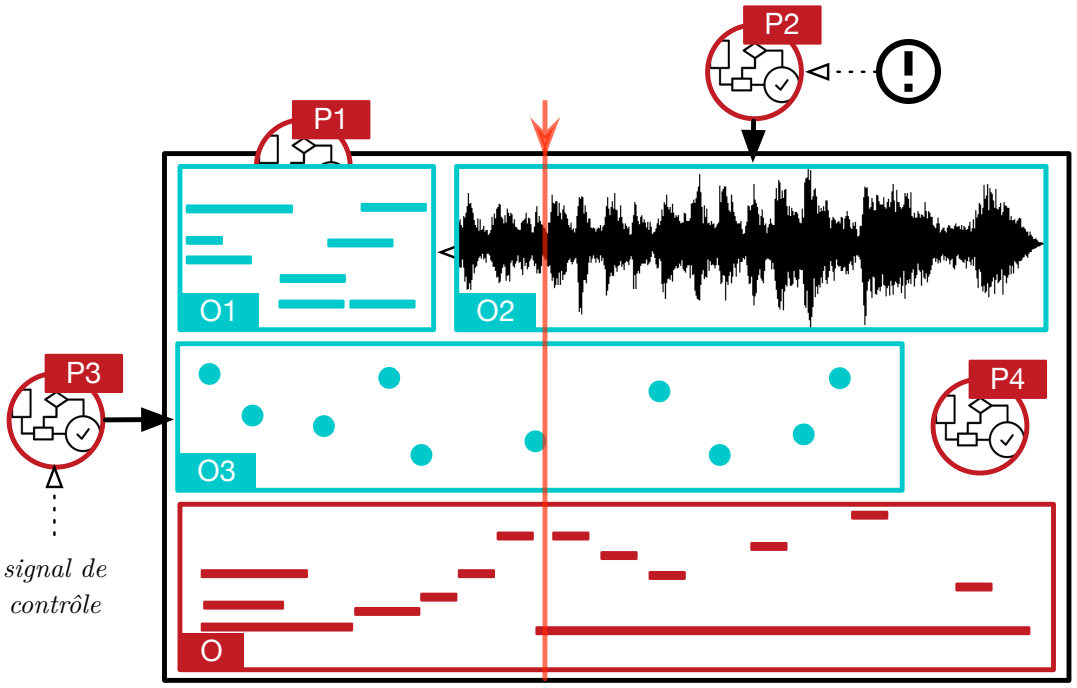


Figure 1.14 – Exemple de meta-composition : état de la partition après la réception d'un événement par P2.

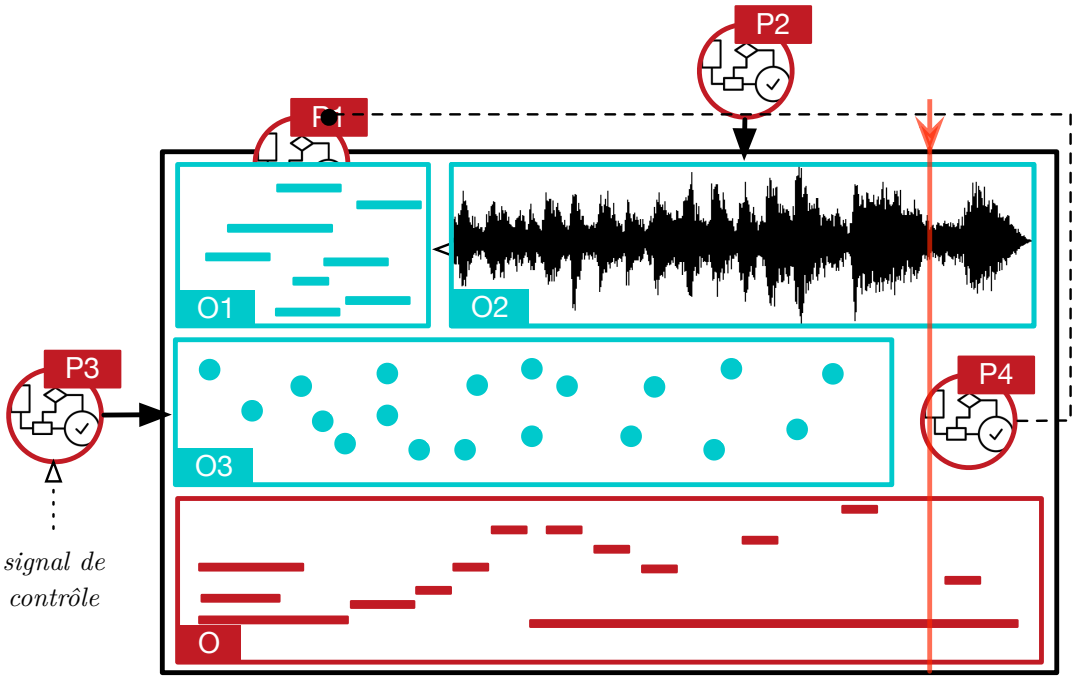


Figure 1.15 – Exemple de meta-composition : état de la partition après l'évaluation de P4.

II

MODÉLISATION DU TEMPS POUR LA REPRÉSENTATION, L'ORDONNANCEMENT ET LA RESTITUTION DES STRUCTURES MUSICALES

Ce chapitre aborde les questions de représentation et de restitution des données musicales.

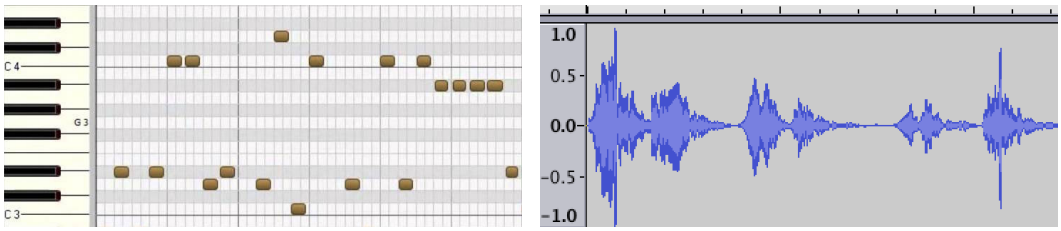
Ici, nous entendons par *représentation* la manière dont les données sont encodées dans les systèmes informatiques, ainsi que les interfaces pour leur manipulation et visualisation.

Nous utilisons le terme *restitution* pour parler de la « mise en temps-réel » de ces données, c'est-à-dire (pour la musique) leur transformation en son. Nous évitons ainsi toute confusion avec les termes « lecture » ou encore « exécution », qui peuvent selon le contexte posséder des significations distinctes (« lecture d'une partition », « exécution d'un calcul »).

2.1 Représentation : partition et contenu

Les outils numériques ont suscité de nouvelles représentations des données musicales. Ainsi, une séquence de notes qui seraient disposées sur une portée dans la notation classique, peut être représentée dans un système informatique sous la forme d'un *piano-roll* (voir figure 2.1a). De la même manière, la représentation d'un signal audio numérique est en général un affichage des valeurs de ce signal dans un espace à deux dimensions appelé *waveform* (« forme d'onde », voir figure 2.1b).

Ces différentes données musicales peuvent être organisées dans l'équivalent informatique de la partition qu'est le *séquenceur*. Celui-ci offre une vision mixte des données : la plupart des logiciels audionumériques permettent de mélanger notation pour instrumentiste et affichage du matériel joué par la machine (forme d'onde pour un fichier



(a) Représentation des notes sous forme de *piano-roll*.

(b) Représentation d'un son sous forme de *waveform*.

FIGURE 2.1 – Exemples de représentations numériques de données musicales.



Figure 2.2 – Capture d'écran du logiciel Logic Pro : illustration de la représentation mixte des données.

audio, *piano-roll* pour une séquence de notes, *etc.*). La figure 2.2 montre comment ces différentes représentations et notations peuvent cohabiter dans un même environnement.

2.1.1 ..... *Temps linéaire, hiérarchique et relations temporelles*

Le temps fait partie intégrante des objets musicaux : sa représentation influe sur les interactions possibles avec la structure musicale. Il existe plusieurs types de représentations : absolue, symbolique, événementielle, chronométrique, discrète, *etc.*

La représentation du temps la plus répandue dans les systèmes multimédia est une représentation *séquentielle*, souvent dénommée *timeline*. Utilisée par la plupart des séquenceurs, elle consiste en la disposition d'objets sur une frise chronologique, assignant une date à chacun d'eux. L'utilisation généralisée de cette représentation tient à sa simplicité, notamment pour le montage d'œuvres. Cette simplicité est également son point faible, car une telle représentation n'admet aucune spécification formelle des contraintes et relations temporelles autre que l'attribution d'une date explicite à chaque activité. Ainsi, une édition manuelle des données sur une timeline entraînera potentiellement une perte de l'ordre précédemment établi. Seule, la représentation séquentielle n'est donc utile que dans le cas où un utilisateur dispose des objets dans le temps, sans expliciter leurs relations.

Une extension de cette représentation est la représentation *hiérarchique* des données. Celle-ci consiste simplement en une agrégation des objets : chaque objet possède sa

propre *timeline*, sur laquelle il est possible de disposer d'autres objets, et ainsi de suite. C'est sur cette notion que repose par exemple le logiciel Boxes (Beurivé, 2000). Dans une représentation hiérarchique, la modification temporelle d'un objet entraîne automatiquement celle des objets qu'il contient. Cependant, les relations entre objets ne peuvent être spécifiées qu'au sein d'une même branche de la hiérarchie.

Pour permettre une meilleure expression des relations entre objets, une représentation logique ou fonctionnelle peut être requise. En ce sens, l'algèbre de Allen (1983) propose par exemple un ensemble de relations logiques permettant l'expression de contraintes temporelles entre des intervalles. Cette algèbre contient 13 relations (7 directes et 6 inverses) comme *before, during, finishes, etc.* qui permettent de créer des dépendances entre les débuts et fins des objets. Le séquenceur i-score (voir section 1.3.2) repose sur cette logique (Desainte-Catherine et Allombert, 2004).

2.1.2 *Précision temporelle d'un système musical*

De la restitution d'une partition à la génération d'un signal audio, les données dans un système musical ne sont pas nécessairement traitées à la même fréquence (voir figure 2.3). Nous en distinguons trois niveaux :

1. les actions ponctuelles de contrôle utilisateur (clic sur une interface, déplacement de la souris, *etc.*) sont sporadiques et d'une précision faible ;
2. une partition peut contenir des objets organisés de manière apériodique, avec une finesse temporelle de l'ordre de la milliseconde [1] ;
3. la granularité du signal audio échantillonné est de l'ordre de la microseconde [2], et les délais inter-échantillons sont constants (échantillonnage périodique).

Dans le cas de contrôle « continu » (mouvement continu d'un potentiomètre par exemple), le taux d'échantillonnage des valeurs est en général de l'ordre de la milliseconde (on parle de « taux de contrôle »). Dans un cas idéal, un système musical traitera toutes les données qu'il manipule à la précision la plus fine, qui est celle de l'échantillon audio. Cependant, cette finesse a un coût en terme d'utilisation des ressources. À titre d'exemple, nous pouvons citer Chuck (Wang, 2008), langage pour la synthèse sonore où contrôle et signal sont traités à la précision de l'échantillon audio, au prix d'un usage intensif du processeur.

Nos travaux se situent dans le domaine des outils dédiés à la composition, et sont donc focalisés sur les aspects structurels et de contrôle (les niveaux 1 et 2 citées précédemment). Nous pourrons limiter la précision du système de meta-composition que

1. *L'effet de précédence* ou *effet Haas* énonce que le délai entre deux sons passe inaperçu pour le cerveau humain pour des valeurs inférieures à 1 ms dans le meilleur des cas (cette durée varie selon la nature du son (Litovsky et al., 1999)).
2. Environ 23 µs pour une fréquence d'échantillonnage de 44100 Hz.

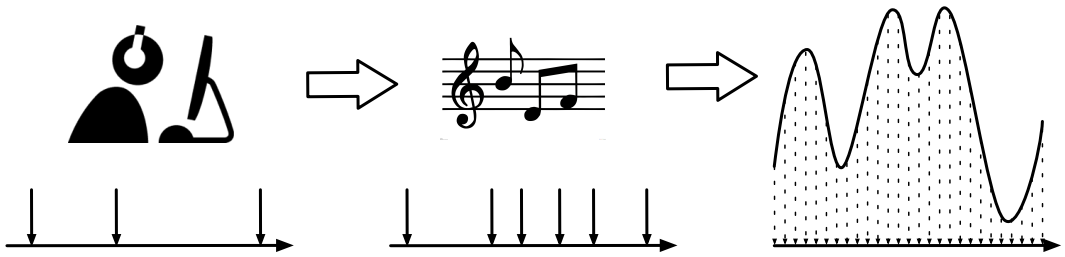


Figure 2.3 – Différentes précisions temporelles d'un système musical.

nous décrivons à la milliseconde. La gestion du signal audio est spécifique et sera détaillée en chapitre 5.

2.2 Modélisation et écoulement du temps

2.2.1 *Système musical discret*

La restitution d'une partition dans un système informatique comprend deux niveaux. Considérons une partition extrêmement simple contenant une seule note. Sa restitution comprend :

1. les événements de déclenchement et d'arrêt de la note,
2. la génération d'un son entre ces deux événements.

Le premier niveau relève du système informatique musical alors que le second peut (ou non) être externalisé. On pourra par exemple utiliser un synthétiseur physique, commandé par l'ordinateur, se chargeant de la production du signal audio. Berry et Cont (2010) décrivent la partition — et la musique en général — comme un mélange d'Automates Finis et de Contrôles Continus. Nous nous limiterons à considérer les systèmes musicaux comme des *systèmes discrets déclenchant des processus continus.*

2.2.2 *Le temps différé de la composition*

Les contraintes auxquelles est soumis un compositeur pendant son travail d'écriture ne sont pas temporelles (au sens de l'écoulement du temps réel). Bien qu'il soit possible d'imaginer un contexte dans lequel une composition doit être livrée dans un intervalle de temps borné, la phase compositionnelle de la création reste généralement la moins contraignante temporellement. C'est durant celle-ci que la majeure partie du travail de réflexion est effectué, y compris dans le cas d'une composition par esquisse, c'est-à-dire par enregistrement et ajustements successifs de phrases musicales.

Nous qualifions de *temps-différé* n'importe quel processus ou système dont :

– le résultat d'un calcul est fourni sans garantie de temps de livraison,
– la temporalité de la restitution des résultats est décorrélée de celle de leurs calculs.

Le premier point décrit une logique d'exécution *best-effort*. L'avantage d'une telle logique est qu'il est possible de mettre en place des processus complexes faisant intervenir des calculs de longues durées pour créer des données qu'il aurait été impossible ou fastidieux de créer manuellement ou en temps réel, ou tout simplement pour automatiser une démarche. Il apparait que ce type de fonctionnement est entièrement adapté à la CAO.

Le deuxième point, bien qu'essentiel au travail de composition, sépare les temps de calcul et d'exécution et laisse donc peu de place à l'interactivité. Il s'agit du principal défi de cette thèse.

2.2.3 *Le temps réel du jeu*

En situation de concert, les contraintes temporelles imposées aux interprètes et aux systèmes informatiques sont extrêmement fortes. Le scénario établi lors de l'écriture doit être respecté de la manière la plus rigoureuse possible, la pertinence musicale résidant en grande partie dans les délais temporels séparants ses éléments.

Le modèle du temps-différé ne concerne que les environnements dans lesquels le temps est exposé au niveau compositionnel. Les environnements temps-réel comme Max n'intègrent pas de représentation explicite du temps et représentent la majorité des éléments constituant leurs programmes comme des générateurs de flux continus. Ils permettent donc de définir des comportements réactifs et fonctionnels, mais la réalisation d'un scénario sera en général externe et viendra contrôler des modules de ces environnements. La figure 2.4 [3] illustre la complexité d'un programme implémentant un *step-sequencer* (séquenceur pas à pas, ne gérant donc que des événements placés à intervalles réguliers) dans l'environnement Max. Comme mentionné en section 1.3.2, PureData offre des moyens de manipuler des structures de données dans le temps, mais l'absence de primitives, de représentation des données musicales conventionnelle ainsi que le modèle du temps utilisé peuvent rendre complexe le processus de composition.

La caractéristique principale des environnements temps-réel est qu'ils entremêlent généralement calcul et restitution : le moteur doit produire des échantillons audio à temps pour qu'ils soient restitués dans l'instant suivant. Une chaîne de calcul trop complexe entrainera l'impossibilité de produire à temps les données nécessaires.

La conséquence d'un dépassement d'échéance pour une tâche d'un système d'informatique musicale peut avoir des conséquences diverses selon sa nature :

3. Illustration extraite de http://audiocookbook.org/step-sequencer-built-in-maxmsp/. Les dernières versions de Max intègrent des primitives permettant un accès simplifié à de telles structures.

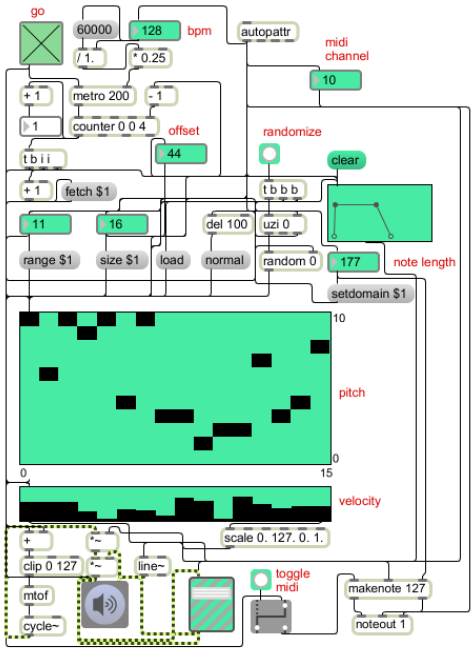


Figure 2.4 – Implémentation d'un *step-sequencer* dans l'environnement Max.

- à l'échelle de la partition : le manquement ou le retard dans le déclenchement d'un événement,
- à l'échelle audio : « clic » ou distortions de la sortie lorsque la carte son ne reçoit pas les échantillons à temps.

Dans Max cependant le respect du temps-réel ne fait pas partie du langage : un dépassement d'échéance aura une conséquence sur la sortie du programme, mais n'est pas considéré comme une erreur d'exécution. La notion de temps-réel peut en effet être affinée en deux sous-catégories que nous décrivons ci-dessous.

temps-réel dur. Aucun dépassement n'est toléré. Un dépassement entraîne une situation critique correspondant à une défaillance ou une panne du système. Le respect des échéances est donc la priorité absolue, quelle que soit la situation dans laquelle le système se trouve.

temps-réel souple. Les dépassements d'échéance sont tolérés dans une certaine mesure au delà de laquelle le système devient inutilisable. Dans ce cas, il est considéré suffisant que les échéances soient respectées la plupart du temps, et que les dépassements puissent être compensés par des avances.

Nous considérons qu'un système musical informatique est une combinaison de temps-réel dur et souple : un dépassement d'échéance ne constitue pas une catastrophe, mais il ne pourra pas être compensé par une avance future et peut constituer une erreur irréparable. Dans la réalisation d'un tel système, les capacités temps-réel doivent être gérées de manière spécifique selon les différents niveaux introduits en section 2.1.2.

Le respect des échéances est donc dépendant de la complexité des programmes créés par l'utilisateur, et non de mécanismes intrinsèques aux logiciels musicaux. Cependant, les contraintes temporelles étant plus fines pour la gestion de l'audio que pour celle des messages de contrôle par exemple, nous verrons que le signal sonore est en général géré par un processus dédié (chapitre 5).

2.2.4 ..... *Paradigmes temporels synchrone et asynchrone*

Nous avons vu en section 2.2.1 qu'un système musical informatique peut être modélisé comme un système discret. Nous introduisons ici les deux paradigmes de programmation des systèmes discrets qui permettront de situer les travaux présentés aux chapitres 3 à 6.

2.2.4.1 . . . . . *Programmation synchrone*

La programmation synchrone est un paradigme réactif. L'abstraction synchrone suppose que les tâches et les communications peuvent s'effectuer simultanément et instantanément : c'est l'hypothèse de *temps-zéro*. La succession des instants constitue le *temps logique*. Dans chaque instant cependant, bien que l'hypothèse synchrone stipule la simultanéité des actions, celles-ci sont par nature ordonnées : un ordre d'exécution respectant la causalité (par exemple lire une variable après l'avoir affectée) est déterminé par le compilateur et peut être vérifié par le programmeur afin qu'il respecte la sémantique de son application.

Lustre (Halbwachs et al., 1991), SIGNAL (Amagbégnon et al., 1995), et Esterel (Berry et Gonthier, 1992) – dont nous donnons une brève description ci-dessous – font partie des langages synchrones les plus populaires.

esterel. Ce langage synchrone impératif et concurrent est très utilisé dans l'aéronautique. L'exécution d'un programme progresse par étape : à chaque instant, le programme calcule ses sorties et son état en fonction des entrées et de l'état précédent. Les tâches concurrentes s'exécutent en communiquant à travers un mécanisme de signaux. À chaque instant un signal est soit présent et permet l'accès à sa valeur, soit absent. Les tâches lisant la valeur d'un signal attendent que les autres aient fixé cette valeur : c'est la relation de causalité qui sert à ordonner les tâches dans l'instant logique. Des méthodes de propagation causale de certitudes sur la présence ou l'absence

de signaux permettent de déduire l'état des autres signaux et de déterminer un ordre d'exécution.

L'hypothèse de temps-zéro énoncée précédemment convient par exemple à des systèmes comme Antescofo, langage synchrone utilisé dans le domaine de l'informatique musicale (introduit en section 1.3.2). En effet, son rôle est centré sur le contrôle d'autres applications, et les calculs produisent essentiellement des messages instantanés. Cette hypothèse doit cependant être vérifiée pour chaque cas d'application : les actions prennent un temps incompressible pouvant entraîner un retard préjudiciable dans l'exécution des programmes.

2.2.4.2 . . . . . *Programmation asynchrone*

Le paradigme asynchrone est en principe plus proche de la réalité. Contrairement au paradigme synchrone, il ne permet pas d'exprimer la simultanéité de deux événements. Deux calculs qui s'effectuent « en parallèle » voient leurs exécutions entrelacées de manière non-déterministe. Comme exemple de langage temps-réel asynchrone, nous pouvons mentionner Electre.

ELECTRE. Electre (Huou et Elloy, 1995) est un langage réactif asynchrone. À la différence des langages synchrones et de leur modélisation discrète du temps, Electre utilise un formalisme inspiré de celui des expressions des chemins (Campbell et Habermann, 1974). Une application temps-réel spécifiée en Electre est découpée en tâches appelées modules, n'intégrant pas de point bloquant. Le langage spécifie l'activation et la préemption des tâches par des événements indépendamment de leur origine, qu'elle soit logicielle ou matérielle. La notion de succession peut être gérée grâce à des événements de terminaison couplés à des activation conditionnelles : l'activation d'un module peut être liée à la fin naturelle d'un autre. Comme dans tout langage asynchrone, deux occurrences d'événements ne sont jamais simultanées.

Le développement d'un programme asynchrone est moins aisé que dans le cas de la programmation synchrone : le programmeur doit prendre en compte l'accès concurrent aux ressources, sans nécessairement en maîtriser l'ordre. Pour pallier ces difficultés, les *Réseaux de Kahn (1974)* proposent une modélisation des systèmes asynchrones comme des ensembles de programmes déterministes (nœuds) communiquant via des files non bornées (arêtes). L'exécution des nœuds est asynchrone, mais la lecture des files étant bloquante, cela permet d'allier programmation asynchrone et déterminisme.

La nécessité de gérer des calculs de durées indéterminées et potentiellement non préemptibles pendant la restitution d'une partition, fait de l'asynchrone une modélisation adaptée à nos objectifs.

> « L'approche asynchrone est celle qui sous-tend les extensions des langages classiques vers le temps-réel. »
>
> (Echeveste, 2015)

2.3 Restitution : mise en temps des structures musicales

Nous appelons *restitution* musicale la transformation de données en musique perceptible. Cette notion peut être considérée comme l'équivalent informatique de l'interprétation d'une œuvre par un musicien. Cependant, à la différence d'un interprète qui modifie certains paramètres d'une œuvre lors de son exécution, le système informatique rend précisément compte des données décrites dans les structures musicales. [4]

La restitution de données statiques ou dynamiques implique une étude des opérations de planification, d'ordonnancement et de répartition d'actions nécessaires à l'exécution d'un système discret. Nous abordons successivement ces notions dans la section suivante.

2.3.1 *Planification*

La *planification* est un domaine de l'informatique ayant pour but de produire des plans d'exécution. On la catégorise comme faisant partie du domaine de l'intelligence artificielle du fait que la construction de plans se fait automatiquement suivant des algorithmes adaptés au contexte particulier de chaque application.

Un planificateur considère les états d'un système et a pour but de le faire progresser d'un état initial à un état cible de manière optimale. Cette optimalité peut être définie suivant des critères différents selon les contextes. Par exemple, un planificateur peut avoir à passer par un état particulier avant d'atteindre son but, ou tout simplement de passer par le moins d'états possibles. Pour produire un plan, un planificateur doit donc pouvoir accéder à :

- une description de l'état initial du système ;
- une description du but à atteindre ;
- des contraintes sur le chemin à emprunter ;
- l'ensemble des actions possibles.

Dans le cas où l'exécution du système est vouée à être modifiée dynamiquement, le plan initial peut être modifié. On parle alors de planification entremêlée à l'exécution, et plus particulièrement de *re-planification* (desJardins et al., 1999).

Une approche pour gérer l'indéterminisme dû à l'exécution d'un programme dynamique consiste en l'énumération de tous les états possibles pouvant apparaître durant

4. L'ordinateur ne remplace donc pas l'interprète, mais peut intégrer de l'indéterminisme dans la restitution (voir par exemple la notion de « partition virtuelle » de Manoury, 2008).

l'exécution, et la prévision d'un plan pour chacun d'eux. On constitue ainsi un ensemble de chemins conditionnels, alternatives à chaque nouvel état issu de l'exécution (Boutilier et al., 1999). L'approche *plan monitoring and repair* (Ambros-Ingerson et Steel, 1988) permet d'éviter cette énumération préliminaire des plans (potentiellement exponentielle) en construisant un unique plan initial qui sera révisé lorsqu'une déviation aux conditions nécessaires est détectée.

Cependant, cette stratégie ne répond pas aux besoins d'environnements complexes, où la déviation aux conditions n'est pas le seul événement devant conduire à une révision du plan. C'est le cas lorsque le contexte d'un programme change d'une manière telle que le plan actuel ne viole pas les conditions fixées, mais permet d'obtenir un nouveau plan optimal (Cohen et Levesque, 1990). La technique de *continual planning* admet cette révision continuelle du plan d'exécution, en considérant que les plans ne devraient être calculés que jusqu'à un certain horizon.

D'autres techniques de planification, plus éloignées de notre problématique, introduisent la planification coopérative entre plusieurs agents (*mixed-initiative planning*), comme la coopération homme/machine par exemple (Ferguson et al., 1999), ou la planification parallèle et distribuée (Wellman, 1992).

2.3.2 ..... *Ordonnancement*

L'*ordonnancement* (Graham et al., 1979) consiste à calculer des dates d'exécutions optimales pour des tâches (calculs de durées non négligeables) ou actions (opérations atomiques de durée négligeable) afin de satisfaire des contraintes temporelles, ou de qualité d'exécution. Cette opération succède généralement celle de planification. Dans le domaine de la planification et de l'ordonnancement automatiques, la notion d'ordonnanceur est plus souvent utilisée pour qualifier un algorithme d'allocation des ressources pour les tâches à effectuer. L'ordonnancement succède à la planification du fait qu'il affecte des dates et alloue des ressources pour les exécutions des nœuds des plans précédemment produits.

Les contraintes d'ordonnancement peuvent être de deux natures :

- contraintes temporelles, que ce soit le respect d'échéances ou de temps moyen d'exécution des calculs ;
- contraintes de ressources, par exemple ne pas avoir plus de n modules réalisant des tâches simultanément, limiter le débit des tâches, limiter leur coût *etc.*

Pour réaliser des ordonnancements optimaux, un ordonnanceur doit connaitre toutes les données du problème auquel il fait face :

- l'ensemble des tâches et actions à réaliser ;
- leurs échéances ;

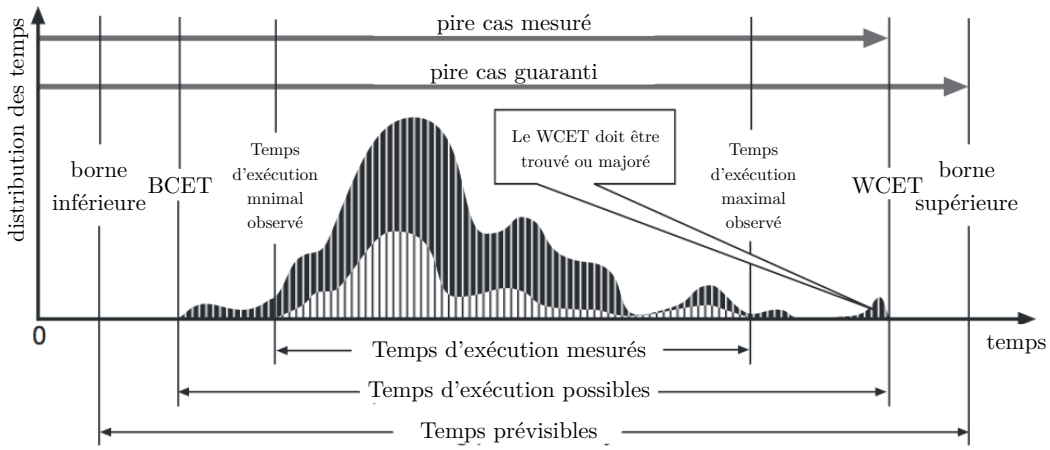


Figure 2.5 – Illustration du WCET, extrait de Wilhelm et al. (2008).

– leurs durées.

Tandis que les deux premiers points sont, en général, facilement obtenus dans le cas de programmes statiques, le dernier est plus problématique. En effet, la durée d'exécution d'un calcul n'est pas constante du fait de l'imperfection matérielle de n'importe quel support physique sur lequel est implanté un système : les environnements physiques et logiciels entrent en compte et perturbent les exécutions. Pour autant, il est possible pour un ordonnanceur de pallier ce problème via l'estimation et l'utilisation de WCET (*Worst Case Execution Time* – Wilhelm et al., 2008).

Le WCET permet de déterminer la limite supérieure du temps d'exécution d'une portion de code donnée (tâche) pour une application particulière en considérant le système physique sur laquelle elle est implantée. Il existe deux classes principales de méthodes de son calcul :

– les méthodes statiques, ne nécessitant aucune exécution du code ni simulation ou expérimentation physique. Ces méthodes parviennent à calculer le WCET en analysant les exécutions bas niveau possibles pour le code étudié, et en combinant ces analyses à un modèle abstrait de l'architecture matérielle sur lequel il est implanté (Wilhelm, 2005) ;
– les méthodes dynamiques (Desikan et al. 2001, Austin et al. 2002) sont basées sur l'exécution répétée du code et l'extraction de la durée d'exécution maximale. Ces méthodes, bien que moins sûres que les premières (il est possible de manquer le pire cas), permettent également d'obtenir des statistiques sur la distribution des temps de calcul, et donc souvent une vision plus proche de la réalité (voir figure 2.5).

2.3.3 ..... *Répartition*

Une fois que les plans sont construits et que des dates d'exécution ont été affectées à leurs actions, leur distribution est possible. La *répartition*, ou encore *dispatching* (Mckay et Wiers, 2003), décrit le fait d'exécuter les plans issus de la planification et de l'ordonnancement préalable. Nous ne nous attarderons pas sur cette notion, car même si elle peut différer d'un système à l'autre, elle est généralement implémentée sous forme d'une simple *boucle de répartition* qui exécute les nœuds des plans séquentiellement, et gère la connexion au temps réel grâce à des minuteurs. Dans le cas d'un système musical, cela peut se modéliser comme le traitement d'une file d'exécution à priorités définies par les temps (Kahrs, 1993) auxquels les actions sont disposées sur la partition.

2.3.4 ..... *Planification et ordonnancement : application aux systèmes musicaux*

PLANIFICATION DANS UN SYSTÈME MUSICAL. Comme nous venons de l'énoncer, la planification est la recherche d'un chemin optimal pour passer d'un état à un autre dans un programme. Bien que nous puissions qualifier d'« état » la trace d'un système musical à un instant donné (par exemple, à un instant t, le système a exécuté les actions musicales $a_1, \ldots, a_n$), le chemin pour passer d'un état à un autre n'est pas décidable par un algorithme mais est inclus dans les structures musicales mêmes. Un système musical doit opérer une planification s'il permet la description fonctionnelle de structures musicales (par exemple « jouer la gamme tempérée ascendante de ré à si »). Pour cette raison, nous n'étudierons pas plus en détail les questions de planifications, et considérons qu'elle correspond à la phase de calcul des structures musicales (par extension, une re-planification équivaut à une modification du contenu d'une structure musicale).

ORDONNANCEMENT DANS UN SYSTÈME MUSICAL. Nous faisons face à deux problèmes qui ne permettent pas d'appliquer directement les résultats de la littérature vus en section 2.3.2. Tout d'abord, nous étudions des systèmes dynamiques : il n'est pas possible de connaitre l'ensemble des données du problème à tout instant (tâches et échéances). Ensuite, les systèmes musicaux sont destinés à être implantés sur des machines grand public, soit sur des systèmes d'exploitations qui agissent comme des boîtes noires en terme d'ordonnancement : il n'est pas possible de maîtriser totalement l'allocation des ressources, les mécanismes de cache, *etc.* Le calcul d'un WCET dans un tel contexte produit un résultat très pessimiste, tandis que sa probabilité d'apparition est significativement faible. Il n'est donc pas utilisable en l'état. En ce sens, des travaux récents s'intéressent au calcul de pWCET (*Probabilistic Worst Case Execution Time*

– Talaboulma et al., 2015), permettant une estimation réaliste de temps d'exécution, et de proposer des ordonnancements probabilistes.

Dans le contexte de notre étude, nous qualifierons d'« ordonnancement » les opérations :

- d'affectation de dates absolues d'exécution aux actions des plans d'actions inhérents aux structures musicales ;
- de gestion de la synchronisation entre les plans d'actions (exécution simultanée de plusieurs structures) ;
- de choix d'ordre d'exécution des différents processus compositionnels.

Nous ferons une distinction entre l'ordonnancement des actions (concernant la sortie musicale du système) et l'ordonnancement des tâches (concernant le calcul de structures musicales).

2.3.4.1 . . . . . *Approches pour l'ordonnancement et la répartition d'actions*

Nous présentons ici trois stratégies possibles d'ordonnancement des actions, en les liant aux systèmes musicaux. Les actions sont des opérations considérées comme atomiques de durées négligeables.

stratégie statique. La stratégie d'exécution statique est celle généralement adoptée dans les systèmes dédiés à la composition, notamment OpenMusic (voir section 1.2). L'ordonnancement des actions est totalement réalisé au préalable de leur exécution. Par exemple, la partition en figure 2.6 est traduite et « aplatie » en un plan (liste d'actions datées) où l'éventuelle structure hiérarchique n'est pas représentée. L'édition du plan produit par l'ordonnanceur n'est pas autorisée durant la répartition, tous les calculs ont été effectués à l'avance et l'unique rôle de la répartition est d'exécuter les actions en temps voulu. Ainsi, toute modification de la structure nécessite un nouvel ordonnancement complet.

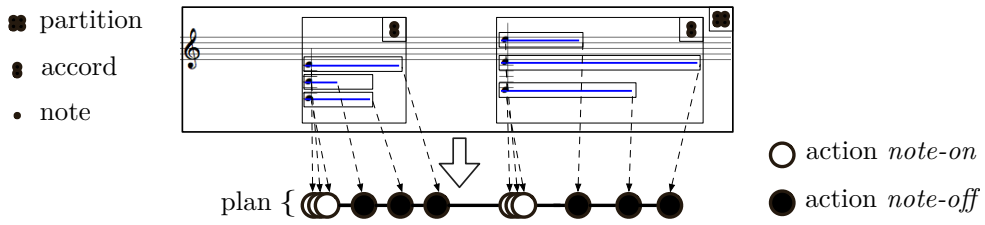


Figure 2.6 – Processus d'ordonnancement statique d'une partition contenant deux accords de trois notes. Les rectangles en traits fins soulignent la hiérarchie des structures musicales, qui n'est pas propagée dans le plan.

stratégie réactive. Les stratégies purement réactives sont utilisées par certains logiciels dédiés à la performance musicale comme Antescofo. À l'opposé de la précédente, celle-ci permet un maximum d'interactions pendant l'exécution d'une structure musicale. L'ordonnancement réactif, largement étudié et utilisé en robotique (Hernandez et Torres, 2013), peut se résumer à ne prévoir qu'une action à la fois. Une fois cette action exécutée, l'ordonnanceur est sollicité pour connaître la prochaine. De cette manière, pendant la restitution d'une partition, n'importe quelle modification peut être opérée pour un coût négligeable, car elle engendre au plus le ré-ordonnancement de la prochaine action. Ce type de stratégie favorise donc les interactions.

En revanche, aucun avantage n'est tiré de la connaissance préalable et sur le long terme que constitue une partition. En particulier, il n'est pas possible de regrouper les calculs de l'ordonnanceur à un instant donné pour optimiser la répartition en temps réduit de nombreuses actions.

Dans ce cas de figure, contrairement à la stratégie statique présentée précédemment, l'ordonnancement est donc entremêlé avec la répartition. Le répartiteur contrôle les appels successifs à l'ordonnanceur qui renvoie des plans *singleton*. La hiérarchie des structures musicales peut ici être préservée : l'exécution d'un objet « parent » peut déclencher celle d'un objet « enfant », qui sera lui aussi sollicité par l'ordonnanceur, et ainsi de suite. L'exemple d'un tel processus d'ordonnancement contrôlé par l'exécution est schématisé dans la figure 2.7.

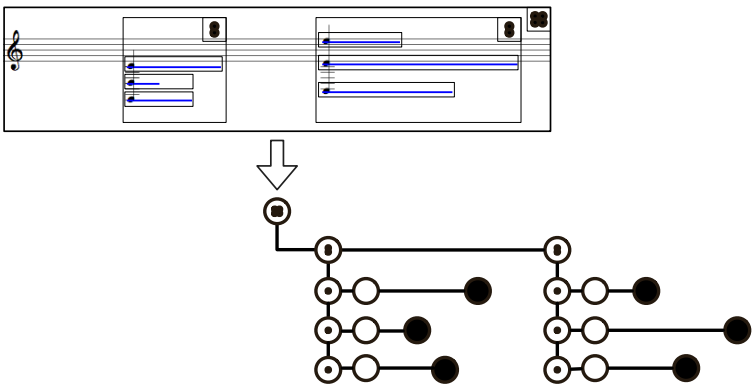


Figure 2.7 – Ordonnancement réactif d'une partition contenant deux accords de trois notes.

stratégies intermédiaires. Notre objectif étant de proposer un environnement adapté à la composition et réactif, nous nous intéressons également à des approches intermédiaires, permettant d'adapter une stratégie statique à des systèmes

dynamiques (Bouche et Bresson, 2015a). Ces approches s'apparentent aux stratégies d'ordonnancement continu. [5]

La première approche consiste à autoriser les modifications sur le plan d'exécution dans une stratégie statique (Bouche et Bresson, 2015b). Si l'on considère que la partition (donc le plan) peut être modifiée pendant son exécution, on autorise ordonnanceur et répartiteur à opérer de manière concurrente, et non plus uniquement séquentielle (Vidal et Nareyek, 2011). L'ordonnanceur fournit un plan complet au préalable mais peut être appelé à tout moment par le répartiteur pour le réviser. [6] Les opérations élémentaires autorisées sur le plan en cours d'exécution sont schématisées en figure 2.8, et sont possibles grâce au maintien d'un lien (par exemple via un pointeur) entre les actions et les objets qui en sont à l'origine : une modification d'un objet peut entraîner la modification du plan en cours de répartition, sans avoir recours à la construction d'un nouveau plan complet.

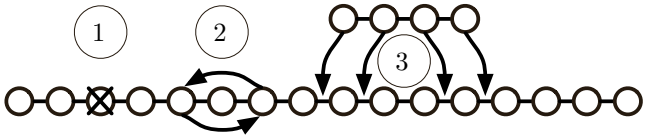


Figure 2.8 – Plan statique éditable : 1) Dé-ordonnancement d'un objet (suppression d'actions) ; 2) Re-ordonnancement d'un objet (déplacement d'actions) ; 3) Ordonnancement d'un objet (ajout d'actions).

Cette stratégie, bien que fonctionnelle, reste inefficace dans le cas de partitions mobilisant dynamiquement un trop grand nombre d'objets. En effet, l'insertion ou suppression d'un objet contenant N actions est réalisée en $\Theta(N)$ et le déplacement en $\mathcal{O}(2N)$. De fait, la répartition du plan peut se trouver mise en attente pendant une durée non négligeable et impacter la cohérence temporelle en sortie.

Considérant la structure hiérarchique d'une partition (conteneur de structures musicales), une seconde approche est d'implémenter un modèle réactif en ne conservant qu'un niveau de hiérarchie (ou un nombre limité de niveaux). [7] La figure 2.9 illustre un ordonnancement à un niveau de hiérarchie. L'ordonnanceur traduit la partition en un « plan d'objets », et chaque objet réfère à un plan qui lui est propre. [8]

5. Similaires au *continual planning* (desJardins et al., 1999), à la différence près que le calcul des plans est, dans notre cas, régi par le compositeur ou par les processus qu'il a créé.

6. À plus bas niveau, l'accès concurrent de ces deux entités sur le plan est sécurisé par un système de verrou.

7. En musique, ne conserver qu'un faible niveau de hiérarchie suffit à un gain important de performances étant donné la profondeur en général limitée des objets inclus dans la partition.

8. Répartir ce plan nécessite donc une double boucle de répartition parcourant le « plan d'objets » et les plans des objets.

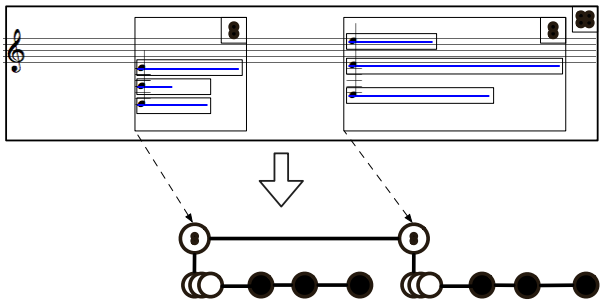


Figure 2.9 – Plan statique éditable à un niveau de hiérarchie.

De cette manière, l'insertion ou l'édition d'un objet dans la partition peut être réalisée en $\mathcal{O}(M)$ où M est le nombre d'objets dans cette partition.[9] En revanche, le problème de complexité de calcul posé par la stratégie précédente perdure (dans une moindre mesure) à un niveau plus bas de la hiérarchie, lors de l'édition du contenu des sous-objets de la partition.

2.3.4.2 . . . . . *Approches pour l'ordonnancement et la répartition des tâches*

En complément de l'ordonnancement des actions, nous citons ici à titre indicatif des algorithmes communs d'ordonnancement des opérations de durées non négligeables (par la suite appelées *tâches* ou *calculs*).

first in, first out. FIFO (Altmeyer et al., 2016) est l'algorithme d'ordonnancement le plus simple. Comme son nom l'indique, il consiste à traiter les tâches dans leur ordre d'arrivée. Il est non préemptif : les tâches en cours d'exécution ne peuvent être interrompues au profit d'autres. De ce fait, son coût est minimal : il ne nécessite aucune réorganisation des tâches, et profite des mécanismes d'optimisation des processeurs (notamment en évitant les changements de contextes pouvant se produire suite à des préemptions). Son débit (nombre de tâches complétées par unité de temps) peut être faible, avec des temps d'attente et de réponse élevés.

earliest deadline first. EDF (Faggioli et al., 2009) est algorithme dynamique d'ordonnancement, préemptif, basé sur un système de priorités. Il permet la complétion prioritaire des tâches dont l'échéance est la plus proche. Pour cela, à chaque complétion d'un tâche ou arrivée d'une nouvelle, il recherche dans la file d'attente la tâche à l'échéance la plus proche dans le temps. S'il existe une tâche en cours d'exécu-

9. Dans un cas d'utilisation musicale réaliste, ce coût se révèle faible en général (quelques centaines au maximum).

tion dont l'échéance est plus lointaine que la nouvelle, elle peut être interrompue pour lui laisser sa place.

round-robin. Dans cet algorithme (Rasmussen et Trick, 2008), une durée unique est assignée à chaque tâche. Chaque tâche peut donc être exécutée pendant cette durée avant d'être interrompue pour laisser place à une autre. L'ordonnancement consiste donc en un cycle entre les tâches. Plus la durée est faible, plus l'algorithme est équitable au prix d'un surcoût (dû aux changements de contexte).

L'algorithme SJF (*Shortest Job First*) (Weber et al., 1986) (que nous ne détaillerons pas) repose sur l'utilisation de WCET (voir section 2.3.2). L'algorithme d'ordonnancement *Fixed priority pre-emptive* est quant à lui similaire au FIFO, mais introduisant un tri de la file d'attente selon un système de priorité, et autorisant la préemption.

L'élaboration d'une stratégie d'ordonnancement des tâches dépend de la problématique considérée. Dans la problématique musicale de meta-composition, les calculs sont intégrés dans la restitution des objets. Les tâches possèdent donc un ordre d'arrivée dans le temps prédéfini. Par conséquent, l'ordonnancement optimal sera en général obtenu grâce à un algorithme simple de type FIFO, comme nous le verrons en section 3.3.

III

SYSTÈME D'ORDONNANCEMENT POUR LA META-CAO

De l'implémentation d'idées musicales sous forme de processus compositionnels à la restitution de données, un environnement de CAO doit réaliser les opérations de calcul, d'ordonnancement et de répartition (voir figure 3.1). Un *calcul* correspond à une évaluation de processus compositionnel, permettant la génération ou la transformation d'objets musicaux. [1] L'opération d'*ordonnancement* consiste en la transformation des données musicales produites en une suite d'actions. Il s'agit de construire des plans datés en temps physique, à partir des plans inhérents aux structures musicales. Finalement, la *répartition* est réalisée par un processus déclenchant les éléments des plans issus de l'ordonnancement en temps voulu, afin de restituer le contenu des structures.

Tandis que ces étapes successives sont communes à la plupart des logiciels musicaux, elles ne sont pas réalisées de la même manière selon le type d'environnement (Bouche et Bresson, 2015b). Dans les environnements de CAO comme OpenMusic, l'ordonnancement est *demand-driven*, et la restitution démarre lorsque le plan est disponible (après les phases de calcul et d'ordonnancement). Dans les environnement temps-réel et de meta-CAO tels que nous les avons décrits en chapitre 1, les phases d'ordonnancement et de restitution sont entremêlées. Des événements — internes ou externes au système — déclenchent des calculs et modifient la sortie du système, ce qui nécessite d'aborder des problématiques de ré-ordonnancement.

Ce chapitre donne en premier lieu une formalisation des données musicales prenant en compte cette intégration des calculs dans la restitution. Il décrit un système permettant d'entremêler calcul, ordonnancement et restitution.

3.1 Formalisation des données

3.1.1 *Actions Musicales*

Une *action* désigne une exécution ponctuelle, par exemple l'envoi d'un message de contrôle à un synthétiseur. Nous considérons une action comme une opération atomique de temps d'exécution négligeable.

ACTION. Soit $(\mathcal{A}, +_{\mathcal{A}}, null)$ l'ensemble des opérations atomiques « instantanées » pouvant être exécutées, avec l'opérateur de composition commutatif et associatif $+_{\mathcal{A}}$.

1. Dans le cas de structures musicales, comme nous l'avons vu en section 2.3.4, la phase de planification est incluse dans le calcul : il n'y a qu'une unique suite d'actions possible pour un objet donné.

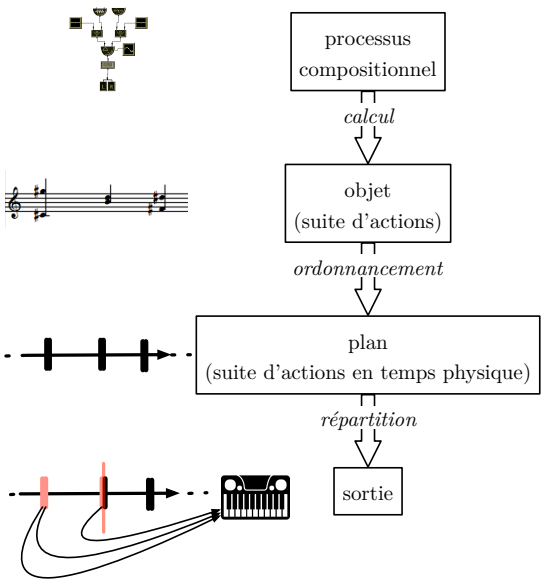


Figure 3.1 – Successions des opérations pour le calcul et la restitution d'objets musicaux.

Grâce aux propriétés de commutativité et d'associativité, une somme d'actions représente une nouvelle action atomique instantanée, correspondant au multi-ensemble d'actions (les termes de la somme) devant être exécutées en parallèle. Une exécution d'action a est nommée un « appel », et sera notée par la suite $call(a)$.

Comme nous l'avons vu en section 2.2.4.1, la programmation synchrone admet la notion de *temps-zéro*, ou d'*instantanéité*. Cette notion théorique permet de simplifier la programmation et de prouver mathématiquement le déroulement de programmes. Nous adoptons cette hypothèse, qui correspond au fait que le résultat de toute action est disponible en temps voulu. Cependant, les calculs étant réalisés en *best-effort* dans un système de CAO, nous n'aurons pas de moyen de garantir sa validité. Nous verrons en section 3.3 comment l'architecture logicielle proposée permet de minimiser l'impact d'une éventuelle violation de cette hypothèse.

3.1.2 ..... *Dates*

Un système musical est fortement relié à l'écoulement du temps : les actions doivent être exécutées dans le bon *ordre* et à la bonne *date*. Une action doit donc être accompagnée d'une information temporelle. Il est important de préciser que les dates ne correspondent pas à des échéances (« exécuter cette action avant telle date ») mais bien à des instant précis (« exécuter cette action à telle date »). Par ailleurs, les dates ne sont pas forcément exprimées en temps physique. Nous considérons donc les dates

comme symboliques, suivant un formalisme arbitraire et opaque à notre système. Par exemple, le plus courant en musique est la combinaison d'un tempo et d'une position comptée en nombre de pulsations.

Nous considérons dans la suite de nos travaux que le temps physique est un entier positif[2], et définissons l'ensemble des dates comme suit.

DATES. Soit $(\mathcal{D}, <_{\mathcal{D}})$ l'ensemble des dates équipé de l'ordre total $<_{\mathcal{D}}$. Une date est une expression caractérisant une information temporelle (symbolique, absolue, relative, *etc.*). Elle peut être convertie en temps physique ($t \in \mathbb{N}$) utilisant l'opérateur d'évaluation

$$eval_C : \mathcal{D} \mapsto \mathbb{N}.$$

Par définition, $<_{\mathcal{D}}$ est défini par : $d <_{\mathcal{D}} d'$ si et seulement si $eval_C(d) < eval_C(d')$. L'indice C représente le contexte dans lequel la date $\in \mathcal{D}$ est évaluée. Par exemple, soit une date d exprimée en nombre de pulsations, relativement à un tempo constant (en battements par seconde) et à partir d'un instant donné. Le contexte C est ici la combinaison du tempo et du temps physique t_{abs} auquel l'évaluation est requise. On a donc

$$eval_{\{t_{abs},tempo\}}(d) = t_{abs} + d \times \frac{60}{tempo}$$

Nous n'explicitons pas la fonction *eval*, étant donné qu'elle peut correspondre à n'importe quel calcul. Nous notons par la suite $t = eval(d)$ la conversion d'une date symbolique d en temps physique.

3.1.3 ..... *Objets musicaux*

Le système que nous décrivons ne fait pas d'hypothèse particulière sur la structure d'un objet, mais lui affecte un certain nombre d'attributs. Un objet est restitué sous la forme d'un plan d'actions, comme définies en section 3.1.1, mais la fonction permettant de transformer le contenu d'un objet en actions datées est extérieure au système et doit être fournie (voir section 4.2.2).

Un objet possède une date relativement à laquelle les dates de chacune de ses actions sont exprimées. Par exemple, un objet *note* de date d_1 correspond *a minima* à deux actions permettant sa restitution par un synthétiseur MIDI : une action *note-on*, et une *note-off*. Dans le référentiel de l'objet, l'action *note-on* est de date zéro et la *note-off* a une date de même valeur que la durée δ de la note. Dans le référentiel contenant la note, l'action *note-on* est de date d_1 et la *note-off* de date $d_1 + \delta$. Cette propriété nous permettra d'organiser les objets dans des structures hiérarchiques.

2. La granularité du système que nous considérons étant la milliseconde (voir section 2.1.2), une date n correspond à n ms.

objets. Soit $\mathcal{O}$ l'ensemble des objets musicaux. Un objet $o \in \mathcal{O}$ est défini comme un quadruplet

$$o = (id_o, d_o, p_o)$$

où :

- id_o est son identifiant ;
- $d_o \in \mathcal{D}$ est la date de l'objet ;
- $p_o(t)$ est une suite d'actions datées correspondant à la restitution de l'objet. Nous appelons cet ensemble le *plan*. Il s'agit d'une fonction de t car il est susceptible d'évoluer au cours du temps.

Pour formaliser la notion de plan d'actions, nous introduisons dans un premier temps l'opérateur associatif et commutatif $+_p$ qui permet l'agrégation de deux plans. Cet opérateur vérifie

$$(d, a_1) +_p (d, a_2) = (d, a_1 +_{\mathcal{A}} a_2). \tag{3.1}$$

Cette propriété exprime le fait que deux actions a_1 et a_2 de même date d équivaut à l'action parallèle $(a_1 +_{\mathcal{A}} a_2)$. Avec cet opérateur, un plan d'actions p à un temps donné t s'écrit sous la forme d'une somme d'actions $p(t) = \sum_i (d_i, a_i)$. Par convention, les termes de la somme sont ordonnés suivant les dates, et donc $d_0 < d_1 < \ldots$

Pour un plan donné, les éléments de la somme peuvent être tous distingués puisque si un même élément est présent de multiples fois, l'équation 3.1 permet de remplacer toutes ses occurrences par un nouvel élément. Autrement dit, un plan est simplement un ensemble d'actions datées. Nous noterons $\mathcal{P}$ l'ensemble des plans.

Nous soulignons que $p(t)$ ne représente pas les actions à effectuer après la date t, mais le plan d'actions correspondant à l'évolution de l'objet au cours du temps, du début du temps de l'objet jusqu'à la fin des temps. En effet, le plan d'actions correspondant à un objet o peut être utilisé comme une donnée dans un calcul compositionnel et peut nécessiter les informations correspondant à toutes les actions de l'objet, indépendamment du temps auquel cette information est utilisée (comme nous le verrons en section 6.1.1).

Pour cette raison, nous aurons besoin d'un opérateur permettant de ne conserver que les actions dont la date d'exécution est supérieure à une date donnée : si $p(t) = \sum_{i=0}^{N} (d_i, a_i)$ alors $p(t)\uparrow_{t'} = \sum_{j=c}^{N} (d_j, a_j)$ avec c le premier indice (le plus petit) tel que $t' \leq d_c$. Notons que $p(t)\uparrow_t$ représente au temps t les actions correspondant au futur de l'objet.

Inversement, on a : si $p(t) = \sum_{i=0}^{N} (d_i, a_i)$ alors $p(t)\downarrow^{t'} = \sum_{j=0}^{c} (d_j, a_j)$ avec c le dernier indice (le plus grand) tel que $d_c < t'$. Notons que $p(t)\downarrow^t$ représente au temps t les actions correspondant au passé de l'objet dans son plan actuel (soulignons que ce n'est pas la trace de la restitution, mais le préfixe d'un certain plan obtenu à la date t).

On définit $\mathit{first}_t(o) = (d_c, a_c)$ si et seulement si $p_o(t)\uparrow_t = \sum_{i=c}^{N}(d_i, a_i)$. Autrement dit, $\mathit{first}_t(o)$ est la connaissance au temps t de la prochaine action datée à exécuter pour restituer o.

On définit $\mathit{last}_t(o) = (d_c, a_c)$ si et seulement si $p_o(t)\downarrow^t = \sum_{i=0}^{c}(d_i, a_i)$. Autrement dit, $\mathit{last}_t(o)$ est la connaissance au temps t de la dernière action datée à exécuter avant t pour restituer o.

3.1.4 *Actions de déclenchements de tâches*

Une *tâche* désigne un calcul entraînant la création ou la modification d'objets musicaux.[3] Le principe de meta-composition permet le déclenchement de tâches dans la restitution des objets (voir l'exemple en section 1.3.2). Donc si, pour un objet o à un instant t on a : $p_o(t)\uparrow_t = \sum_{i=0}^{N}(d_i, a_i)$ et que a_0 est une tâche, alors $p_o(t) \neq p_o(t_0)$ (on suppose ici qu'une tâche modifie toujours le plan d'actions et prend place instantanément). Par la suite, nous noterons parfois une action qui déclenche une tâche par le symbole a_c.

3.1.5 *Trace de restitution*

La trace d'un objet est la représentation de sa restitution. On peut donc la modéliser par un élément de $\mathcal{P}$ où les dates correspondent à des temps physiques et où les actions dénotent leurs propres exécutions. On appellera ces traces $\mathcal{T} = \mathbb{N} \times \mathcal{A}$. Les opérations définies sur $\mathcal{P}$ agissent aussi sur $\mathcal{T}$ puisque $\mathcal{T} \subset \mathcal{P}$.

3.2 Mécanismes d'exécution

Nous présentons dans cette section les « mécanismes d'exécution » du système, c'est à dire les opérations d'ordonnancement et de restitution des objets musicaux.

ordonnancement d'un objet. La sémantique de la restitution d'un objet o est définie par la trace $[\![o]\!]$ de ses actions :

$$\begin{aligned} [\![\cdot]\!] : \mathcal{O} &\mapsto \mathcal{T} \\ (id_o, d_o, p_o) &\to [\![o]\!]_0 \end{aligned}$$

avec

$$[\![(id_o, d_o, p_o)]\!]_t = \big(eval(d_o) + eval(d_0), a_0\big) + [\![o]\!]_{t_0} \quad \text{si } p_o(t)\uparrow_t = \sum_{i=0}^{N}(d_i, a_i)\,.$$

3. Dans le cadre de la meta-CAO, une tâche correspond à l'évaluation d'un processus compositionnel.

Cette dernière définition décrit l'ordonnancement des actions d'un objet à partir d'une date t : d_o est la date de l'objet et d_0 la date de la première action du plan de l'objet à exécuter après t.

RESTITUTION DES PLANS. La restitution des plans issus de l'ordonnancement est réalisée dans une *boucle de répartition* permettant l'exécution en temps voulu des actions contenues dans les objets ordonnancés. La boucle de répartition sélectionne la prochaine action à exécuter dans le plan d'un objet, puis se place en attente jusqu'à atteindre sa date physique, afin de l'exécuter. L'algorithme 1 décrit ce processus.

L'attente intègre en réalité un comportement actif, permettant de réagir à toute modification opérée sur le plan d'un objet (lors de la requête de restitution d'un objet, ou d'une modification de son contenu), auquel cas la prochaine action à exécuter est mise à jour.

Algorithm 1 Boucle de répartition d'un objet o

$t \leftarrow 0$

while *true* **do**
  $(t', a) \leftarrow \mathit{first}_t(o)$
  $\mathit{wait}(t' - t)$
  $t \leftarrow t'$
  $\mathit{call}(a)$
end while

3.3 Optimisation pour la meta-composition

Dans la formalisation des actions en section 3.1.1, nous avons admis l'hypothèse synchrone, à savoir l'instantanéité de leurs exécutions y compris pour les actions déclenchant des tâches. Cependant, ces tâches peuvent être de durée non négligeables : l'évaluation de processus compositionnels est de complexité arbitraire. En ce sens, nous concevons notre architecture en séparant les processus (*threads*) de calcul, d'ordonnancement et de restitution de telle manière que l'impact de leurs opérations les uns sur les autres soit limité et proposons un certain nombre de mécanismes permettant d'assurer le bon déroulement et la continuité dans la restitution d'objets musicaux. L'implémentation de l'architecture est schématisée dans la figure 3.2, que nous détaillons dans cette section.

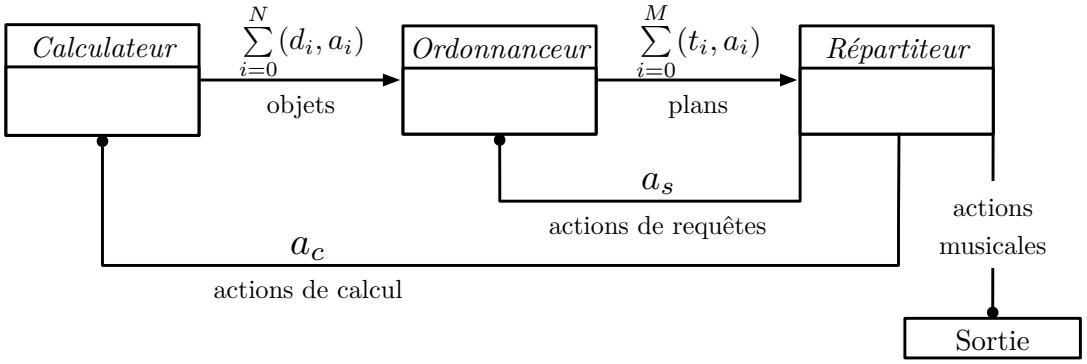


Figure 3.2 – Architecture complète.

calculateur. Le processus de calcul, appelé *calculateur*, est en charge de fournir des résultats en exécutant des tâches (évaluations de processus compositionnels). Ces tâches peuvent provenir de requêtes d'utilisateur, d'événements extérieurs ou de commandes déclenchées par la restitution (actions de calcul, voir section 3.1.4). [4]

Les tâches sont placées dans une file d'attente et exécutées en séquence. Cette file est triée par dates d'échéance croissantes. Ce comportement se différencie de l'algorithme classique Earliest Deadline First car il est non préemptif, et permet simplement de favoriser la complétion des tâches avant leur date butoir : c'est en réalité une stratégie intermédiaire entre FIFO et Fixed priority (voir section 2.3.4.2). Comme nous le verrons en section 6.3, l'utilisateur pourra influer sur cet ordonnancement en précisant au système d'exécuter une tâche avec un certain temps d'avance.

Le calculateur peut être implémenté comme un processus unique, ou bien comme un groupe de processus. La deuxième possibilité nécessite simplement un mécanisme de répartition équitable des tâches entre les processus (appelé *thread pool*, Kriemann, 2003). Elle ouvre également la voie à des algorithmes d'ordonnancement des tâches plus sophistiqués, notamment préemptifs. Cependant, dans la problématique musicale que nous considérons, l'ordre d'arrivée des tâches dans le calculateur correspond en général à l'ordre souhaité en sortie. Dans un tel cas de figure, une exécution séquentielle, ou tout au plus parallèle, est souvent préférable à une succession de préemptions.

ordonnanceur. Le processus d'ordonnancement, appelé *ordonnanceur*, est en charge de fournir des plans d'actions à partir d'objets. Il possède un registre pour stocker les références de chaque objet devant être restitué. Des objets sont inscrits dans

4. Dans cette section, toute interaction avec l'environnement est modélisée par des objets spéciaux qui changent de plan « spontanément », contrairement aux objets dont le plan change à la suite d'une action de calcul.

le registre automatiquement par le système [5] ou lorsque l'utilisateur le demande (par exemple lors du démarrage de la restitution d'un objet par une commande « play »).

Ordonnancer un objet n'est pas nécessairement réalisé instantanément : la fonction produisant des actions à partir du contenu d'un objet est opaque à notre système, et la durée de cette opération est en pratique proportionnelle au nombre d'actions de la structure musicale.

Réaliser l'ordonnancement des objets en une passe n'est en général pas nécessaire, et fragmenter cette opération permet d'être plus réactifs aux changements de plans. Si l'on ordonnance un objet action par action, il est possible de s'abstraire de tout mécanisme de ré-ordonnancement au prix d'un nombre de requêtes maximal (cela correspond à la stratégie d'ordonnancement réactif vu en section 2.3.4.1). En ce sens, nous développons dans cette section un algorithme d'ordonnancement flexible : il sera possible selon le contexte d'ordonnancer un objet action par action, en une passe, ou suivant un comportement adaptatif que nous détaillons.

Ordonnancement adaptatif. Nous définissons la sémantique d'ordonnancement « borné », permettant de calculer le plan d'un objet suivant un *horizon* temporel, par :

$$\begin{aligned} [\![\cdot]\!]_t^\delta : \mathcal{O} &\mapsto \mathcal{T} \\ (id_o, d_o, p_o) &\to \sum_{i=i_0}^{i_n} (eval(d_o) + eval(d_i), a_i) \end{aligned} \tag{3.2}$$

avec $p_o(t)\uparrow_t = \sum_{i=i_0}^{N}(d_i, a_i)$ et $p_o(t)\downarrow^{t+\delta} = \sum_{i=0}^{i_n}(d_i, a_i)$.

Fragmenter les opérations d'ordonnancement par horizons successifs requiert un mécanisme de requêtes d'ordonnancement automatiques. Cette opération n'étant pas considérée comme instantanée, l'ordonnanceur doit toujours prévoir un plan d'avance afin de ne pas retarder la restitution. Pour cela, l'ordonnanceur et le répartiteur fonctionnent en parallèle : lorsque que le répartiteur reçoit un plan de l'ordonnanceur, il démarre sa restitution pendant que le prochain plan est préparé. Nous intégrons donc systématiquement au début des plans des objets une action déclenchant une requête d'ordonnancement (atomique et instantanée) pour le plan suivant, que l'on note a_s. On peut donc qualifier l'ordonnancement d'incrémental et autonome, et dont la sémantique devient

$$\begin{aligned} [\![\cdot]\!]_t^\delta : \mathcal{O} &\mapsto \mathcal{T} \\ (id_o, d_o, p_o) &\to (eval(d_o) + t, a_s(o)) + \sum_{i=i_0}^{i_n} (eval(d_o) + eval(d_i), a_i) \end{aligned}$$

5. Nous verrons par la suite que la restitution d'un objet peut par exemple déclencher celle d'un sous-objet.

avec les conditions identiques à l'équation 3.2 et $a_s(o)$ l'action de déclenchement de l'ordonnancement du plan suivant.

La durée des plans produits (l'horizon δ) est contrôlée suivant un comportement adaptatif. L'ordonnanceur va procéder à une série d'ordonnancements successifs, produisant des *sous-plans* correspondant à des portions d'objets définis par des horizons variables. Par exemple, dans le cas d'un objet o « statique », c'est à dire qui n'est pas voué à être modifié par des calculs, on peut noter $[\![o]\!]_t^\delta = [\![o]\!]_t$, soit $\delta = \infty$. Dans le cas d'un objet déclenchant des calculs réguliers suivant une période T, on préfèrera $\delta = T$. Lorsque les calculs sont répartis de manière arbitraire dans un objet, l'ordonnancement peut progresser étape par étape, avec des horizons variables (par exemple définis par la date de la prochaine action de calcul à exécuter).

Si nous écrivons $\delta_1, \delta_2, \ldots$ les horizons d'ordonnancement successifs et $\Delta[\![o]\!]_t^\delta$ la durée d'une opération d'ordonnancement $[\![o]\!]_t^\delta$, la condition suivante doit rester vraie afin de ne pas subir de retard dans la production des plans d'actions :

$$\forall k \in \mathbb{N}^*, \quad \Delta[\![o]\!]_t^{\delta_k} < \delta_{k-1}. \tag{3.3}$$

Nous définissons par défaut cette progression des horizons empiriquement afin qu'ils croissent suffisamment rapidement pour limiter le nombre d'opérations d'ordonnancement, et suffisamment lentement pour préserver la condition précédente (eq. 3.3) vraie. Elle est actuellement implémentée comme la progression géométrique

$$\begin{aligned} \delta_0 &= \delta_{min}, \\ \delta_{k+1} &= \delta_k * r, \\ r &\in]1; +\infty[\end{aligned} \tag{3.4}$$

où δ_{min} est la durée la plus faible possible pour un plan, et r le ratio de la progression géométrique. [6]

Cette progression générique, illustrée en figure 3.3, se positionne à l'intermédiaire des stratégies d'ordonnancement statique et réactive (voir section 2.3.4.1). L'intérêt de réaliser un ordonnancement court (proche de la stratégie réactive) au départ est d'éviter un trop grand retard au démarrage d'une restitution. L'augmentation de sa longueur permet au système de tendre vers un comportement statique. La progression peut cependant être définie différemment, et non pas forcément comme une suite au sens mathématique du terme. Il est possible de définir manuellement chaque horizon, sans nécessairement établir de relation formelle entre δ_k et δ_{k+1}, comme nous en donnerons un exemple en section 4.2.2.

Ré-ordonnancement conditionnel. Le modèle précédent ne prend pas en compte les interactions utilisateur, qui peuvent également venir modifier les objets en cours de

6. En pratique, notre système utilise actuellement $\delta_{min} = 10$ ms et $r = 2$.

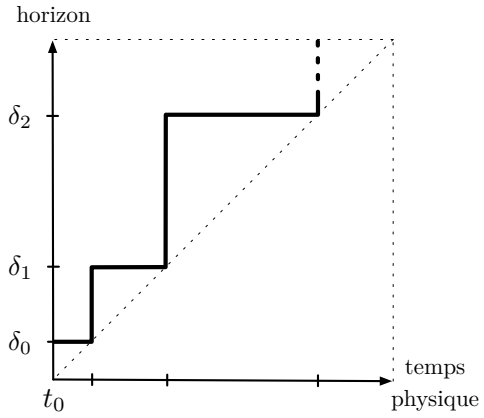


Figure 3.3 – Progression générique des horizons d'ordonnancement.

restitution. Il fait cependant l'hypothèse que la probabilité d'un objet à être modifié par une interaction est décroissante au cours du temps, à partir de la dernière occurence. Plus le temps s'écoule dans le système sans qu'une modification soit notifiée, plus il est probable qu'un ordonnancement de long terme soit valide. La progression de l'horizon par défaut, qui croît progressivement tandis que l'objet n'est pas modifié, permet d'économiser les ressources allouées aux opérations d'ordonnancement pendant les portions « statiques » de restitution.

En revanche, pour être réactif aux modifications d'objets suite à des interactions, l'ordonnanceur doit rester actif durant la restitution afin d'opérer des ré-ordonnancements si nécessaire. Si le calculateur termine au temps physique t_c un calcul qui modifie un objet pendant qu'un de ses sous-plans est en cours de restitution, il notifie l'ordonnanceur qui décide, selon la partie de l'objet affectée et l'horizon courant, si une opération de ré-ordonnancement est nécessaire. Nous appelons cette stratégie un ré-ordonnancement conditionnel (voir algorithme 2). Dans ce cas, la progression est réinitialisée, présumant que de nouvelles modifications sont susceptibles d'avoir lieu par la suite.

Algorithm 2 Ré-ordonnancement conditionnel

$p = \sum_{i=0}^{n}(t_i, a_i) \leftarrow$ sous-plan de l'objet o en cours de restitution
$t_c \leftarrow$ date physique de fin de calcul
$\mathcal{I}_M \leftarrow$ intervalle de modification de l'objet en temps physique
$p_/ = p\uparrow_{t_c+\delta_{min}}$

if $([t_0; t_n] \cap \mathcal{I}_M \neq \emptyset) \wedge (t_n - t_0 > \delta_{min})$ **then**
  $p \leftarrow p_/ + [\![o]\!]_{t_c+\delta_{min}}^{\delta_{min}}$
end if

Cette architecture réactive permet un calcul rapide de nouveaux plans quand le système est perturbé : les modifications sont prises en compte dans la restitution avec un retard fixe de $\delta_{\min}$ ms. Ensuite, la progression d'intervalles démarre à nouveau afin de réduire le nombre d'opérations d'ordonnancement. La figure 3.4 illustre la progression générique des intervalles lorsqu'un ré-ordonnancement intervient.

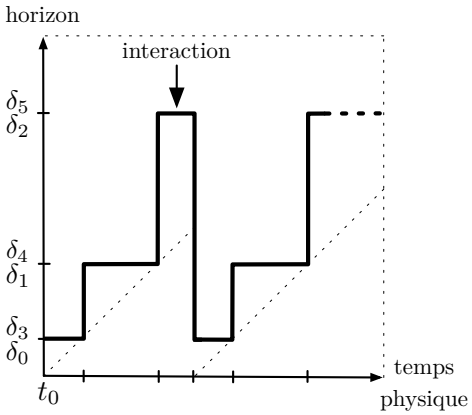


Figure 3.4 – Progression des horizons d'ordonnancement en présence d'une interaction.

La figure 3.5 montre un exemple de déroulement temporel des trois composants principaux de l'architecture pour la lecture d'un objet, avec une modification « interactive » de celui-ci. L'objet est ordonnancé à partir du temps physique t_0 en (1), qui produit un premier plan partiel d'horizon δ_0 que le répartiteur commence à restituer en (2) (dès que le plan est calculé). Pendant l'écoulement de du temps sur $[t_0;\ t_0+\delta_0]$, l'ordonnanceur prépare un deuxième plan, plus long, pour être restitué durant $[t_0+\delta_0;\ t_0+\sum_{i=0}^{1}\delta_i]$, et ainsi de suite. Une tâche est déclenchée dans le calculateur en (3) (par exemple depuis un contrôle utilisateur). En (4), au milieu de l'écoulement de $[t_0+\sum_{i=0}^{1}\delta_i;\ t_0+\sum_{i=0}^{2}\delta_i]$, ce calcul termine : l'ordonnanceur réinitialise la progression en générant un nouveau plan à court terme de durée δ_0.

⋆ ⋆ ⋆

L'architecture décrite dans ce chapitre, composée des modules de calcul, d'ordonnancement et de répartition, est implantée dans le logiciel OpenMusic sous la forme de trois *threads* concurrents interconnectés. Comme nous le verrons dans le chapitre suivant, elle est accompagnée d'une structure de données permettant l'implémentation d'objets musicaux adaptés.

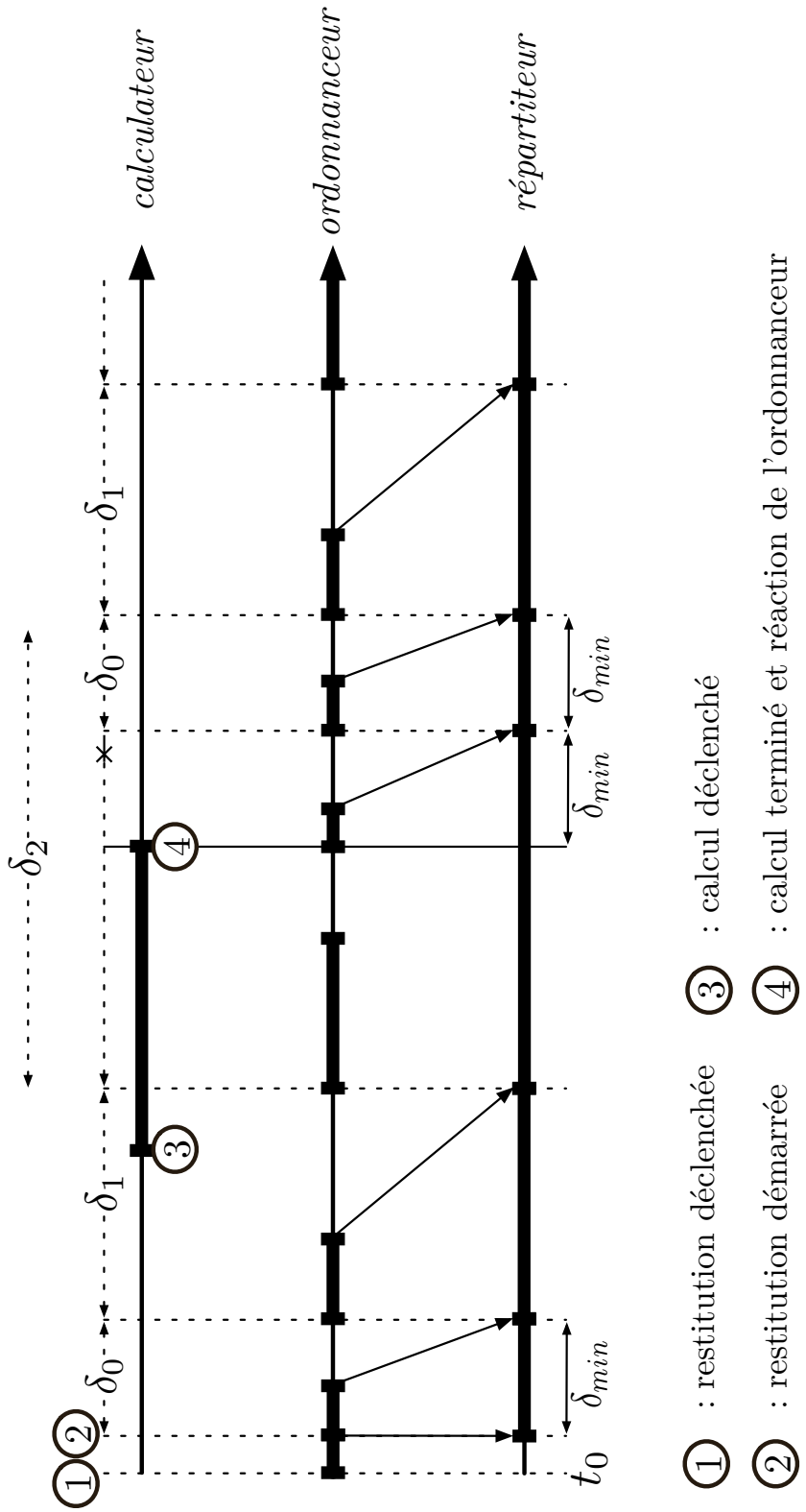


Figure 3.5 – Exemple de déroulement temporel des processus de l'architecture.

IV

IMPLÉMENTATION : UNE ARCHITECTURE POUR LA REPRÉSENTATION ET LA RESTITUTION DYNAMIQUE DES OBJETS MUSICAUX

Le système d'ordonnancement présenté dans le chapitre précédent permet de restituer indifféremment les objets musicaux et les structures composées : il peut être utilisé comme un « player », mais permet aussi de réaliser des applications interactives, comme nous le verrons dans les chapitres suivants.

Nous décrivons dans ce chapitre l'implémentation d'une structure de donnée permettant aux objet musicaux, dans le sens de la formalisation vue en section 3.1.3, d'être développés et gérés par notre système d'ordonnancement. La classe définie est nommée s-object (pour *schedulable-object*), et ses caractéristiques peuvent être classées en deux catégories : celle dédiée à la représentation/manipulation des données par l'utilisateur [1], et celle dédiée à leur restitution (voir figure 4.1).

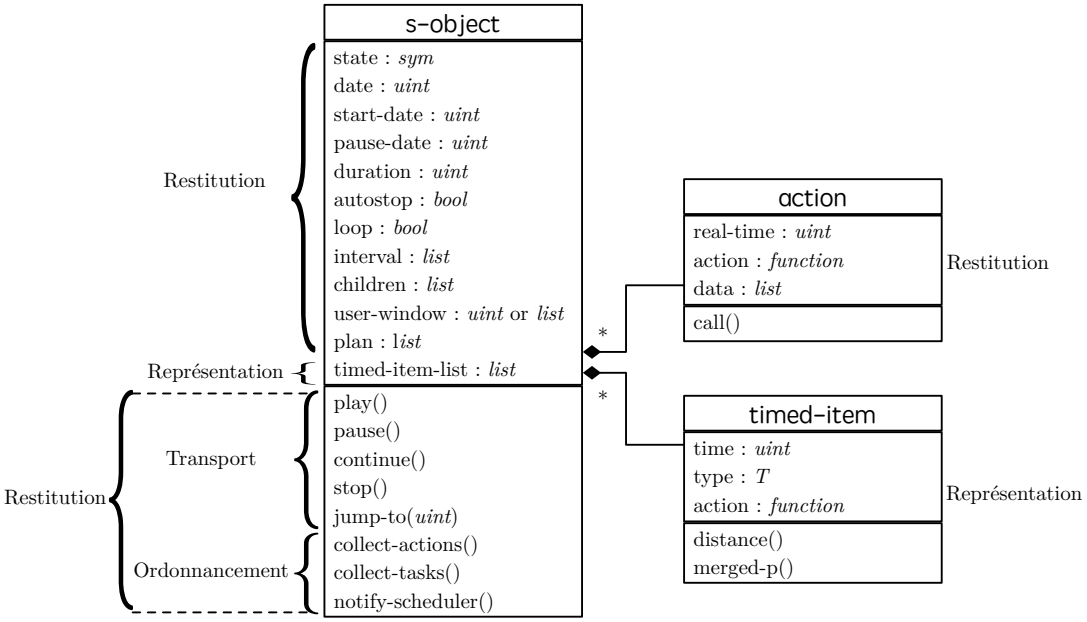


FIGURE 4.1 – Classes s-object et timed-item. Les accesseurs/mutateurs ne sont pas représentés.

1. Caractéristiques développées en collaboration avec Jérémie Garcia — http://jeremiegarcia.fr/

4.1 Représentation et manipulation des objets musicaux

4.1.1 *Interface de programmation*

Afin de permettre la manipulation d'objets au sens de la formalisation vue en section 3.1.3, la définition de la classe s-object inclut une collection d'éléments datés, appelés timed-items. Les sous-classes de s-object doivent alors contenir et pouvoir renvoyer un ensemble d'instances de timed-items (ou d'une autre structure possédant le même ensemble d'accesseurs publics).

- get-timed-item-list : $\mathbb{N}^2 \mapsto \{\text{timed-item}\}^*$
  $\Rightarrow$ *obtenir la liste de timed-items d'un objet entre deux dates.*
- set-timed-item-list : $\{\text{timed-item}\}^* \mapsto$ s-object
  $\Rightarrow$ *définir la liste de timed-items d'un objet.*

Chaque timed-item possède une valeur de temps, pouvant être « explicite » (donnée par l'utilisateur) ou « implicite ». Cette différenciation tient au fait qu'il n'est pas toujours nécessaire à un timed-item d'avoir une date associée. [2] Lorsque la date d'un élément est implicite, celui-ci doit être inclus dans une séquence où il existe un élément daté explicitement avant et après lui. Le calcul de la date par le système est alors possible grâce à une mesure de distance entre deux timed-items qui permet une interpolation du temps.

La classe timed-item différencie les catégories d'éléments avec des rôles et comportements spécifiques via un attribut de *type* $\in T = \{timed;\ untimed;\ master\}$:

- *timed* pour les timed-items dont la date explicite,
- *untimed* pour ceux dont la date est implicite,
- *master* pour ceux dont la date est explicite, utilisés comme des points d'ancrage pour des transformations ou synchronisations de s-objects (voir section 6.4.2). [3]

4.1.2 *Éditeur graphique et opérations*

Un composant graphique de type *timeline* est associé à la classe s-object, permettant la visualisation des éléments datés et l'application de transformations temporelles. Il

2. Par exemple, si l'on considère une trajectoire dans l'espace (pour un processus de spatialisation sonore), un compositeur peut souhaiter en définir une durée globale, sans assigner une date à chaque point de la trajectoire. Chaque point possède donc une date, qui peut être soit définie par l'utilisateur, soit calculée automatiquement par le système.

3. Les premiers et derniers éléments de la liste de timed-items sont toujours considérés comme *master* et ont donc une date assignée par défaut. Cela permet la consistance du modèle et satisfait les besoins classique d'étirement, de compression ou de synchronisation des objets.

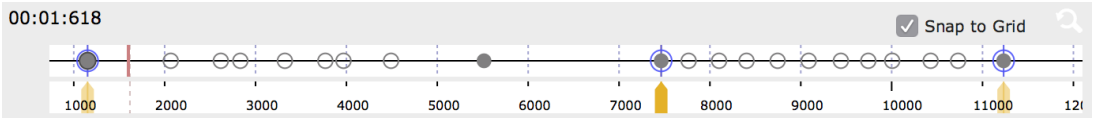


Figure 4.2 – Composant graphique associé à la structure temporelle d'un s-object. Les timed-items de différents types sont représentés respectivement par ∘ (*untimed*), • (*timed*) et ⊙ (*master*).

peut être intégré dans un éditeur graphique plus complexe pour gérer l'interaction et l'affichage d'information temporelle. La figure 4.2 en montre un exemple.

La selection, l'ajout et la suppression de timed-items sont possibles et propagent l'information dans le s-object correspondant. L'affectation et la modification de dates sont principalement effectuées manuellement (avec la souris) par des translations sur la *timeline*. Des raccourcis (clavier) permettent la modification du type des timed-items.

Selon son *type*, les effets de l'affectation d'une date à un timed-item varient :

- dater un timed-item *untimed* change son type à *timed*,
- modifier la date d'un timed-item *timed* met à jour la date de ses voisins (directs et indirects) de type *untimed*,
- modifier la date d'un timed-item *master* modifie la date de ses voisins *timed* (directs et indirects), et modifie également par propagation la date de ses voisins (directs et indirects) de type *untimed*. [4]

La figure 4.3 illustre différents scénarios.

4.2 Restitution des objets musicaux

4.2.1 *Méthodes de transport*

La classe s-object est reliée à notre système d'ordonnancement via une interface de programmation (API) simple permettant le contrôle de la restitution. Cette API présente les méthodes play/pause/continue/stop (généralement qualifiées de « transport »). Ces méthodes contrôlent l'inscription d'un s-object au registre de l'ordonnanceur et en affectent l'attribut state afin que l'environnement ait connaissance de son état (restitution en cours, en pause ou stoppée). Le lien de la restitution du s-object au temps réel est possible grâce à ses attributs start-date et pause-date qui permettent à l'ordonnanceur de lui affecter une date physique de démarrage et éventuellement de pause

4. L'ensemble des types T est par conséquent strictement ordonné, ce qui introduit une hiérarchie dans la structure temporelle qui influence la manière dont les dates explicites ou dates internes sont calculées : la date d'un timed-item est systématiquement mise à jour selon la date de ses plus proches voisins d'ordre supérieur.

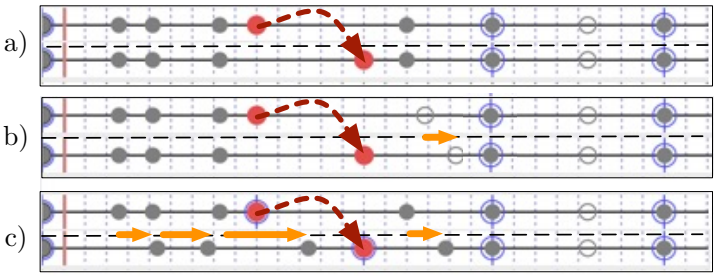


Figure 4.3 – Conséquence d'une modification de date dans un *timeline-editor* (les flèches oranges illustrent les transformations consécutives à la modification) :
a) Simple translation d'un timed-item *timed* (avec deux voisins *timed*).
b) Translation avec mise à jour d'un timed-item *untimed* (à droite) pour préserver un ratio constant.
c) Étirement temporel entre deux timed-items *master*.

de la restitution, afin d'être en mesure de calculer son temps à chaque instant (en comparant ces attributs à l'heure du système). Un appel à la méthode play déclenche la première requête d'ordonnancement. Nous rappelons ici que les requêtes d'ordonnancement sont intégrées aux plans par l'ordonnanceur lui-même (voir section 3.2). Par conséquent, seule la première requête d'ordonnancement doit être appelée explicitement. La gestion de la restitution est complétée par une méthode permettant de changer le temps de l'objet pendant sa restitution :

– jump-to : $\mathbb{N} \mapsto$ s-object
  $\Rightarrow$ *change la date de restitution actuelle.*

Finalement, pour fournir un contrôle complet de la restitution, il est possible de préciser si la restitution doit s'arrêter seule une fois le contenu de l'objet épuisé. Cette précision se fait via l'attribut autostop. Dans le cas où celui ci est activé, le système à besoin de récupérer la durée d'un objet, via l'attribut duration. Cette option est utile dans le cas où un objet « vide » ou incomplet est voué à être alimenté dynamiquement. En pratique, la prévision de l'arrêt automatique de la restitution est réalisée par l'ordonnanceur : à chaque plan qu'il construit, il vérifie si la durée de l'objet est atteinte et prévoit (ou non) l'arrêt de sa restitution en conséquence (retrait du registre).

Finalement, l'attribut loop indique à l'ordonnanceur si celui-ci doit « boucler » la restitution, et l'attribut interval contrôle l'intervalle de restitution. Ces informations sont vérifiées à chaque passe d'ordonnancement afin d'en ajuster les bornes, et de prévoir ou non le redémarrage de la restitution à une date donnée.

4.2.2 *Méthodes d'ordonnancement*

Les actions des plans sont définies dans une structure interne au système, contenant une expression à évaluer et les données qu'elle requiert si elle déclenche une tâche, afin d'être répartie vers le calculateur le cas échéant.

Pour permettre à l'ordonnanceur de collecter les plans des objets afin de les lier au temps réel, et de différencier les actions déclenchant des calculs des actions musicales « instantanées », l'interface de programmation est complétée par deux méthodes d'ordonnancement qui doivent être redéfinies pour chaque classe héritant de la classe s-object :

- collect-tasks : $\mathbb{N}^2 \mapsto list$
  $\Rightarrow$ *renvoie une liste de tâches à exécuter pour un intervalle donné.*
- collect-actions : $\mathbb{N}^2 \mapsto list$
  $\Rightarrow$ *renvoie une liste d'actions à exécuter pour un intervalle donné.*

Ces méthodes sont appelées successivement par l'ordonnanceur. Les résultats obtenus sont assemblés en un unique plan stocké grâce à l'attribut plan. Par défaut, ces méthodes se reportent sur l'attribut action des timed-items situés dans l'intervalle d'ordonnancement. Nous verrons par la suite qu'elle peuvent être surchargées pour implémenter des comportements plus avancés (voir en section 4.3).

HIÉRARCHIE DES OBJETS. Notre système ne fait pas d'hypothèse sur la nature du contenu des objets qu'il manipule. Un objet peut en contenir d'autres, mais le système décrit en section 3.1.3 considère toujours les plans comme des « mises à plat » des contenus. En règle générale, les méthodes d'ordonnancement précédentes s'appelleront donc récursivement pour chaque objet contenu dans un autre (nous parlerons alors de « restitution simple »). [5]

Il est cependant possible de répercuter la hiérarchie d'un objet dans son ordonnancement grâce à la mise en parallèle des restitutions de ses enfants. Pour cela, un plan d'action renvoyé par la méthode collect-actions pourra contenir des appels datés à la méthode play introduite en section 4.2.1, qui déclenchera l'ordonnancement de sous-objets. Ces appels sont des actions au sens de la définition donnée en section 3.1.1 car, comme nous l'avons vu en section 3.3, le processus de restitution s'exécute en parallèle du processus d'ordonnancement. Lorsque que la restitution d'un objet parent déclenche celle d'un objet enfant, celui-ci est inscrit au registre de l'ordonnanceur et dans l'attribut children de son objet parent. Cet attribut permet de répercuter les changements d'état d'un objet parent sur ses enfants en cours de restitution. Si la méthode collect-actions

5. Nous considérons un objet *racine* qui est en pratique le seul a être restitué par notre système. Son plan est initialement vide et sera modifié à chaque requête de restitution d'un autre objet afin d'intégrer les actions ordonnancées.

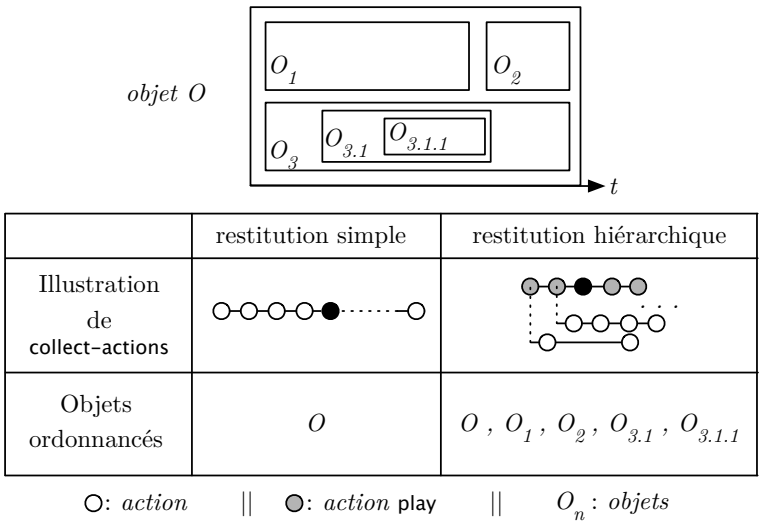


Figure 4.4 – Illustration de la différence entre une restitution simple et hiérarchique.

est implémentée de la sorte, nous pourrons parler de « restitution hiérarchique » (voir figure 4.4).

Afin de supporter ces deux modes d'ordonnancement, la boucle de répartition vue en algorithme 1 chapitre 3 opère en réalité sur une liste d'objets ordonnancés en parallèle.

horizons d'ordonnancement. Les méthodes d'ordonnancement possèdent comme paramètres deux bornes temporelles définies par la date de départ et l'horizon δ_k d'ordonnancement en cours. Ces horizons sont propres à chaque objet et définis, par défaut, comme une progression gérée automatiquement par le système (voir section 3.3). Prenons l'exemple d'un objet devant appeler périodiquement une action : il sera dans ce cas certainement optimal de fixer l'horizon à la valeur de la période. Également, si le contenu d'un objet est voué à être régulièrement modifié seulement pour une portion de sa restitution, la progression d'horizons optimale peut être définie manuellement. Par exemple, si seule la deuxième moitié d'un objet est vouée à être régulièrement modifiée durant sa restitution, et que l'on note d sa durée, la progression permettant de s'abstraire de toute opération de ré-ordonnancement serait

$$\delta_0 = \frac{d}{2},$$
$$\forall n > 0,\ \delta_n = \delta_{min}.$$

Pour cela, nous donnons la possibilité de préciser un horizon particulier ou une liste d'horizons représentant une progression grâce à l'attribut user-window.

réactivité et ré-ordonnancement. Les interactions extérieures venant modifier le contenu d'un objet peuvent être signalées au système grâce à la méthode

– notify-scheduler : $\mathbb{N}^2 \mapsto \{true, false\}$
  $\Rightarrow$ *informe l'ordonnanceur d'une modification de contenu.*

L'intervalle temporel d'une modification d'un objet (dans son référentiel) est un argument optionnel, qui permet d'optimiser le ré-ordonnancement conditionnel vu en algorithme 2 (section 3.2). Si l'intervalle n'est pas précisé, l'algorithme déclenche par défaut un ré-ordonnancement. Cette méthode renvoie un booléen précisant si un ré-ordonnancement est déclenché ou non.

Les actions d'un plan renvoyé par la méthode collect-tasks sont également susceptibles de modifier le contenu des objets. Tout comme les actions, les tâches sont renvoyées avec une date d'exécution, qui correspond donc un changement potentiel dans la structure musicale. L'ordonnanceur peut ainsi ajuster automatiquement l'horizon temporel courant en fonction de la date de prochaine tâche. [6]

Nous donnons le diagramme de séquence de la restitution d'un s-object simple en annexe B.1, et celui d'un s-object dont la restitution déclenche un calcul en annexe B.2.

4.3 Exemples d'implémentation

Nous donnons en annexe A un modèle générique de code permettant de connecter une classe d'objet avec l'architecture présentée par l'héritage de la classe s-object. Nous donnons ici quelques exemples d'objets musicaux implémentés dans l'environnement OpenMusic :

– La bpf, l'automation, la bpc sont des outils de contrôle permettant de tracer des fonctions ou des courbes, et de paramétrer les actions correspondant à leur restitution. Une bpf représente une fonction sous forme d'une succession de points. Une automation est une version de la bpf échantillonnée dédiée au contrôle « continu » de paramètres et dont l'interpolation entre deux points peut-être paramétrée, d'un comportement logarithmique à exponentiel. La bpc est similaire à la bpf mais permet de tracer des courbes bi-dimensionnelles : l'abscisse des points ne représente plus le temps. Finalement, la 3dc est une extension tri-dimensionnelle de l'objet bpc.
– Le data-stream et le piano-roll sont des outils dédiés à l'agrégation de messages de contrôle datés. Un data-stream est une simple séquence temporelle de « paquets » d'actions (plusieurs dans le même instant) à réaliser. Typiquement, cet objet peut être utilisé afin d'écrire un scenario temporel constitué d'envois de messages. Le

6. Une anticipation de la date d'exécution d'une tâche nous permettra, dans l'implémentation du système, de compenser sa durée d'exécution (voir sections 5.4 et 6.3).

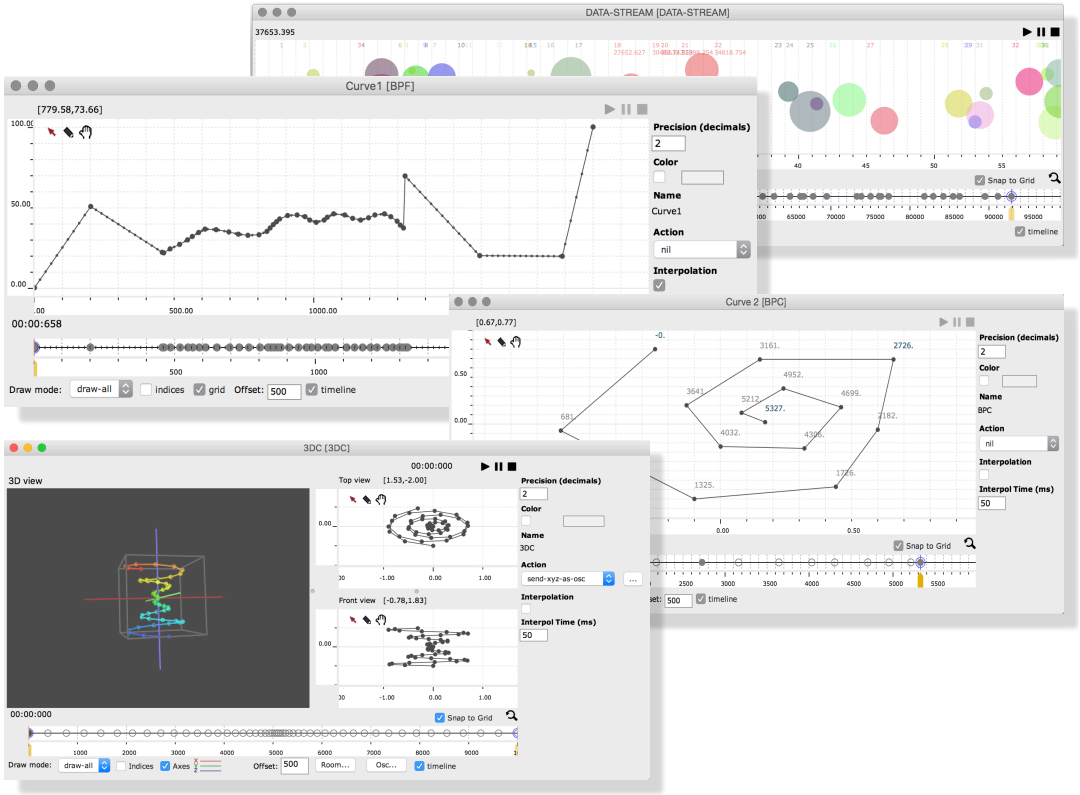


Figure 4.5 – Objets de contrôle implémentés à partir de la structure s-object (incluant une représentation de la structure temporelle). De haut en bas : data-stream, bpf, bpc, et 3dc.

piano-roll est une spécialisation du data-stream dédié à l'envoi de messages MIDI de type *note-on*/*note-off*.

Les figures 4.5 et 4.6 montrent les éditeurs de ces différents objets auxquels sont intégrés le composant graphique décrit en section 4.1.2, qui permet d'uniformiser leur représentation temporelle.

Spécialisation des actions. Chaque s-object spécialise ses méthodes collect-actions et collect-tasks. La plupart des objets implémentés ci-dessus définissent ou permettent à l'utilisateur de définir une fonction associée à la restitution d'un type de timed-item donné (dans les cas les plus fréquents, l'envoi de messages MIDI, OSC, *etc.*).

Les méthodes d'ordonnancement peuvent également intégrer des mécanismes plus élaborés pour la collecte d'actions et de tâches, y compris décorrelés de la séquence de timed-items. Par exemple, il est possible d'implémenter des fonctions d'interpolation

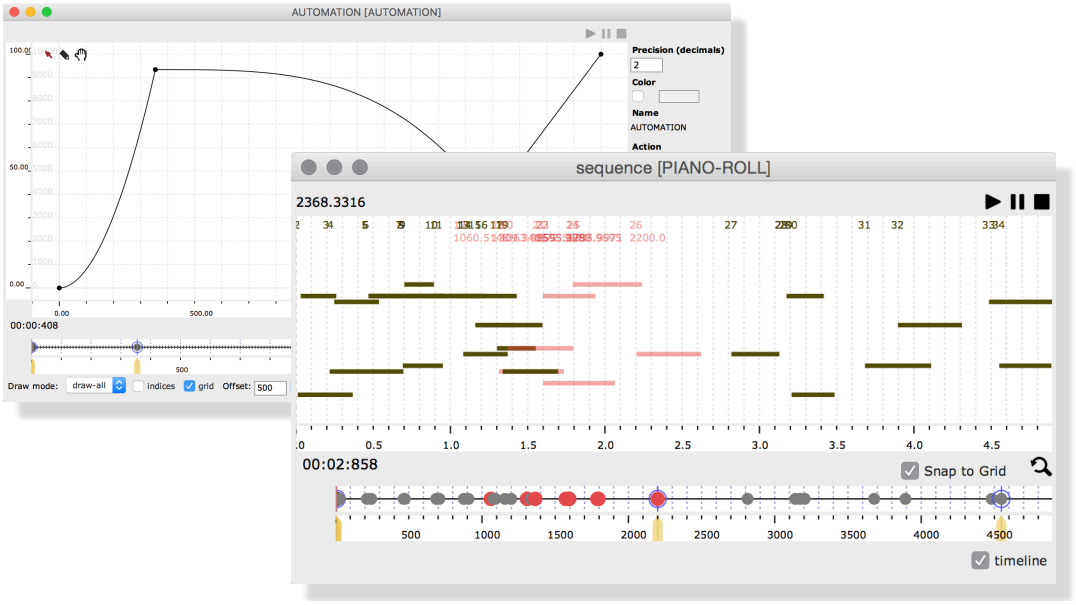


Figure 4.6 – Objets musicaux dédiés au MIDI implémentés à partir de la structure s-object : automation et piano-roll.

pour fournir une séquence périodique d'actions et/ou de tâches sur l'intervalle d'ordonnancement donné. Nous exploitons cette stratégie pour la restitution d'automations, de trajectoires ou de contrôleurs continus.

⋆ ⋆ ⋆

La structure s-object est particulièrement adaptée aux objets de « contrôle », dont les actions de restitution correspondent à l'émission de messages de commande à destination de synthétiseurs, d'autres logiciels *etc.* Cependant, nous n'avons ici pas abordé la gestion du signal audio, qui répond à des contraintes différentes. Ce point sera l'objet du chapitre suivant.

V

GESTION DES RESSOURCES ET DE LA RESTITUTION SONORE

Jusqu'ici, nous avons considéré notre système musical comme discret, composé de processus qui interagissent de manière asynchrone, et possédant une granularité de l'ordre de la milliseconde. Cependant, la gestion et la restitution du signal sonore demandent une attention particulière : les mécanismes bas niveau doivent être synchrones avec une précision plus importante que celle du système global (voir section 2.1.2). Ce chapitre présente les aspects généraux relatifs aux ressources audio en section 5.1. Cette section présente également l'exemple de l'implémentation du moteur audio d'OpenMusic. Ensuite, la section 5.2 montre comment temps-différé et temps-réel peuvent s'articuler dans la gestion du signal sonore. La section 5.3 présente le moteur audio conçu pour notre système, et nous donnons des exemples d'implémentation d'objets sonores dynamiques en section 5.4.

5.1 Gestion des ressources audio

5.1.1 *Généralités*

La numérisation du signal audio consiste en son échantillonnage par un ADC [1]. Une fois numérisé, le signal est enregistré dans un tableau d'échantillons (voir section 2.1) dont la taille, pour une durée donnée, est dépendante de la fréquence d'échantillonnage et de la précision spécifiée par le nombre de bits sur lequel il est encodé (de 8 à 32bits). Les fréquences d'échantillonnage audio typiques vont de 44.1 kHz (limite inférieure fixée par le théorème de Shannon [2]) à 192 kHz pour la très haute définition.

Procéder à des opérations en temps réel sur un signal numérisé peut donc entraîner un grand nombre d'opérations par unité de temps (44100 opérations par seconde pour une fréquence de 44.1 kHz). De plus, de nombreuses méthodes du traitement du signal nécessitent la considération de multiples échantillons sur une fenêtre temporelle. [3] Les systèmes audio opèrent donc généralement des calculs par blocs d'échantillons, nommés *buffers*. Les calculs effectués peuvent ainsi bénéficier de différentes optimisations du compilateur et du matériel (parallélisation de boucles, vectorisation des accès mémoire, réduction des interruptions, *pipeline*, *etc.*).

Ce mécanisme de traitements par blocs ne concerne pas seulement les calculs, mais également la restitution du signal (comme illustré en figure 5.1). Cette restitution est

1. *Analog to Digital Converter* (convertisseur analogique-numérique).
2. Les fréquences audibles vont jusqu'à 20kHz environ.
3. Notamment afin d'opérer des traitements dans le domaine spectral.

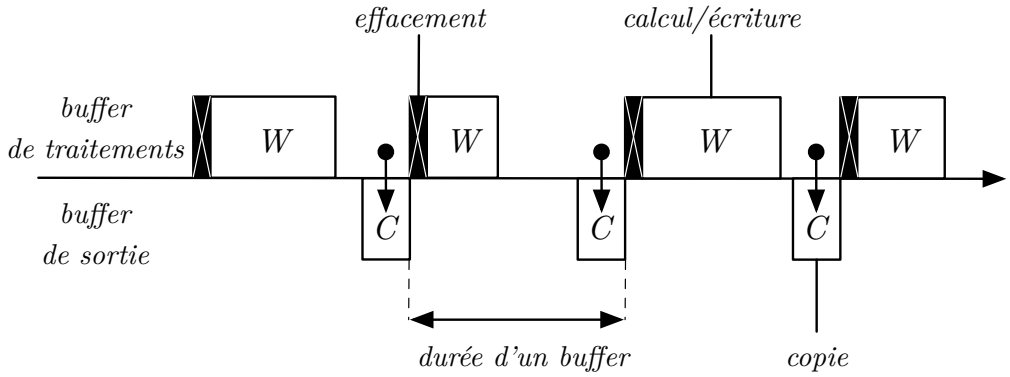


Figure 5.1 – Schématisation de la restitution audio par buffer. Les phases d'écriture W intègrent potentiellement le calcul des échantillons (pour les environnement temps-réel).

guidée par la carte son, qui envoie des requêtes périodique au système afin de collecter des échantillons audio. Entre deux appels de la carte son, le système doit donc copier les échantillons dans le buffer de sortie, mais bien souvent aussi les calculer (dans le cas de systèmes de synthèse ou de traitement audio numérique). Ces opérations sont de durées variables selon la complexité de la chaîne de calcul : la restitution audio peut être mise en échec dans le cas de calculs trop complexes. Certains systèmes adoptent un mécanisme de *multiple buffering* afin d'avoir un plus grand nombre de d'échantillons prêts à l'avance et de pouvoir effectuer une rotation entre les différents buffers.

Introduire une gestion de l'audio sous forme de buffer entraîne aussi une limitation au niveau du contrôle : la modification d'un paramètre durant le traitement d'un buffer ne peut être prise en compte que pour le traitement du buffer suivant.

5.1.2 ..... *Gestion du signal audio dans OpenMusic*

La calcul en temps-différé permet de réaliser des opérations de traitement et de synthèse sonore dont la complexité n'est pas limitée (Bresson, 2007). Dans OpenMusic, les processus de calcul audio sont séparés du processus de restitution : les échantillons audio n'existent pas en tant que donnée éphémère à transmettre à la carte son, mais sous forme de structures de données en mémoire. Cette décorrelation entre calcul et restitution se retrouve dans l'architecture audio de l'environnement (Bouche, 2013) où la restitution audio est confiée à un processus séparé via la bibliothèque temps-réel *LibAudioStream*.

libaudiostream. LibAudioStream (LAS) (Letz et al., 2014) est un moteur de restitution audionumérique disponible sous forme de bibliothèque de fonctions, permettant de manipuler des ressources audio à travers le concept de *flux*. Une algèbre de description, de composition et de transformation des flux audio permet de construire des expressions complexes, ordonnancées à des dates dans le futur et dont la restitution sera déclenchée au moment venu.

Bien que le moteur de LAS soit synchrone, la gestion du contrôle dans cette bibliothèque peut être qualifiée *d'asynchrone*. Dans les systèmes audio classiques, les valeurs des signaux de contrôle restent constantes sur la durée du buffer audio sur lequel elles s'appliquent. Dans LibAudioStream, la datation des événements étant à l'échantillon près, les valeurs peuvent changer à l'intérieur d'un bloc audio : la fonction de restitution découpe le bloc audio en tranches correspondant à des valeurs constantes des paramètres de contrôle.

5.2 Vers une mixité des temps

La séparation entre la capacité de calcul offerte par le temps différé et les fortes contraintes temps-réel de la gestion du signal audio contribue à la catégorisation des environnements d'informatique musicale en temps-différé et temps-réel. Cependant, allier les deux permettrait la génération et le contrôle temps-réel du signal, tout en préservant des constructions structurelles. Nous montrons au fil de cette section que cela est possible au sein d'une architecture unifiée et dynamique.

5.2.1 *Approche GALS*

La restitution du signal audio doit répondre à des contraintes temporelles strictes : il faut que les prochains échantillons soient disponibles à chaque activation périodique de la carte son (période de l'ordre de la milliseconde [4]). L'utilisation de mécanismes synchrones pour une telle tâche périodique est privilégiée, car elle permet de modéliser chaque activation de la carte son comme un instant logique.

Cependant, cette approche est difficilement conciliable avec la présence de calculs erratiques (résultant d'interactions imprévisibles) car elle nécessite soit que les calculs puissent être effectués sans entraîner de retard dans l'avancement du programme, soit qu'ils puissent être interrompus afin d'être continués dans l'instant suivant (au prix d'un mécanisme plus complexe). Nous ne souhaitons faire aucune hypothèse sur les calculs à exécuter par notre système (comme dans les outils de cao en général), et ne pouvons donc pas nous baser sur un mécanisme globalement synchrone.

4. Pour une fréquence d'échantillonnage de 44.1 kHz et une taille de buffer de 64 échantillons.

Ces deux contraintes ne doivent pas forcément s'entremêler. L'établissement d'un modèle de communication et d'échange des données entre un module synchrone et un module asynchrone permet de lier de manière cohérente le domaine temps-différé (du symbolique, de l'interaction utilisateur) au domaine du temps-réel (du signal). C'est le principe du modèle GALS *(Globally Asynchronous Locally Synchronous)* (Carlsson et al., 2006).

GALS. Le modèle de calcul globalement asynchrone / localement synchrone date des années 1980. Celui-ci permet l'agencement de nœuds synchrones (horloges locales), communiquant de manière asynchrone. Le modèle GALS trouve ses applications dans de nombreux domaines, notamment les systèmes temps critique comme l'avionique, ou encore le nucléaire. La vérification formelle de tels systèmes est un domaine de recherche actif (Ramesh et al., 2004).

Notre architecture audio est conçue sur cette idée : le système synchrone interne à notre moteur de restitution audio LAS sera alimenté de manière asynchrone par des données audio.

5.2.2 ..... *L'exemple OM-Faust*

FAUST. *Functional AUdio STream* (Orlarey et al., 2009) est un langage de programmation fonctionnel pour la conception d'applications de traitement du signal. L'objectif de ce langage est d'offrir une notation fonctionnelle pour décrire générateurs et transformateurs de signaux d'un point de vue mathématique, se détachant ainsi le plus possible de tout détail d'implémentation. Les programmes ainsi créés sont entièrement compilés : le compilateur traduit le code Faust en code C++ autonome (Scaringella et al., 2003), qui peut ensuite être compilé pour des environnements cibles spécifiques (Max, PureData, VST *etc.*) (Fober et al., 2011). Un programme écrit en Faust peut préciser quels paramètres sont contrôlés par un utilisateur, afin de produire l'interface de contrôle correpsondante.

Les flux audio dans LibAudioStream peuvent être transformés en temps-réel par des effet décrits en Faust. La compilation de code Faust est réalisée grâce à *libfaust* (Orlarey, 2014), la version dynamique du compilateur Faust associée à LLVM (Lattner et Adve, 2004), qui permet de traduire les traitements synchrones écrits en Faust en code binaire exécutable fonctionnant à vitesse native.

OM-Faust (Bouche et al., 2014) est une bibliothèque destinée à l'environnement OpenMusic permettant d'écrire et compiler et des programmes Faust, contrôlables en temps-réel via LibAudioStream. Le langage Faust peut être intégré de manière

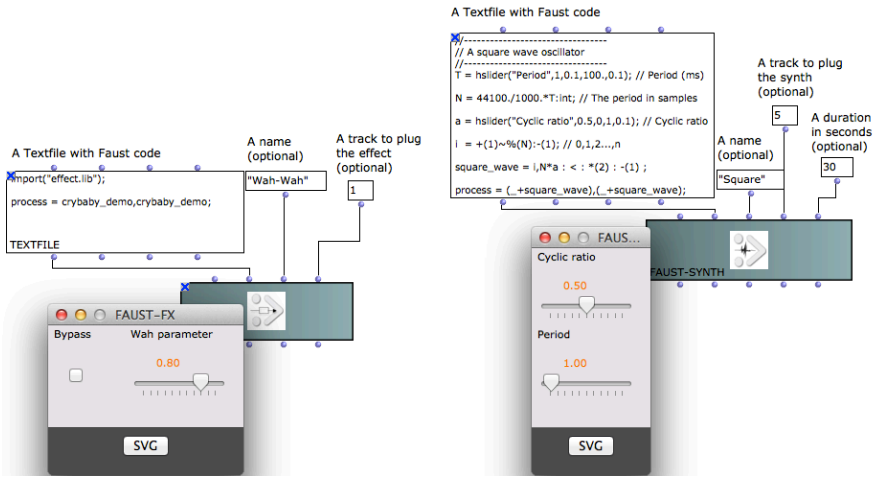


Figure 5.2 – Exemples d'objets faust-fx et faust-synth dans OM-Faust.

dynamique (compilation à la volée et contrôle temps-réel de flux audio) grâce à LibAudioStream et la technologie LLVM. Nous avons conçu cette bibliothèque pour à la fois contrôler en temps-réel des effets et synthétiseurs programmés en Faust, ainsi que composer en temps-différé des sons et de leurs processus de contrôle.

Pour ce faire, elle fournit des objets d'effet (faust-fx) et de synthèse (faust-synth), qui permettent de compiler dynamiquement du code source Faust, et génèrent l'interface de contrôle correspondante (le code Faust permet de préciser quelles variables sont contrôlées et la forme du contrôleur). La figure 5.2 montre un exemple de ces deux objets et des interfaces automatiquement générées par OM-Faust. [5]

Contrôle et audio temps-différé. OM-Faust peut être utilisée au sein d'un processus compositionnel temps-différé. Pour cela, elle présente les méthodes flatten-faust-synth et flatten-faust-fx qui permettent respectivement de produire un fichier son à partir d'un objet de synthèse faust-synth, et d'appliquer un effet faust-fx à un fichier audio afin d'en produire une version transformée (voir figure 5.4). Celles-ci collectent les échantillons que les objets produisent sans les lier au temps physique. On peut parler de « restitution différée », car les objets produisent des données le plus rapidement possible et hors du système de restitution audio.

Contrôle et audio temps-réel. La modification de paramètres via les interfaces de contrôle des faust-synth et faust-fx déclenche des appels en temps-réel à LibAudioStream afin de les répercuter sur les modules Faust compilés. Cela permet

5. Les figures de cette section sont issues de notre publication : Bouche et al. (2014).

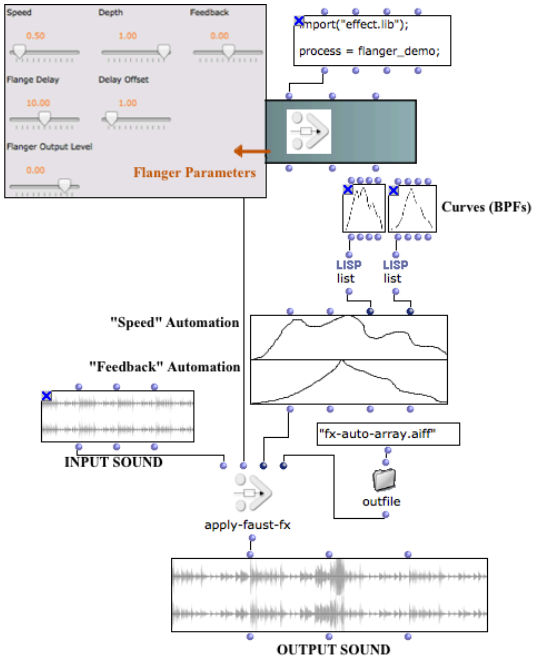


Figure 5.3 – Exemple d'utilisation de la méthode flatten-faust-fx dans OM-Faust.

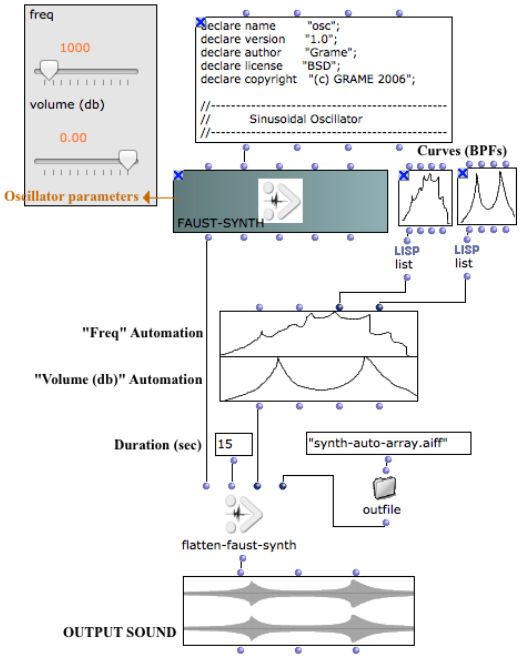


Figure 5.4 – Exemple d'utilisation de la méthode flatten-faust-synth dans OM-Faust.

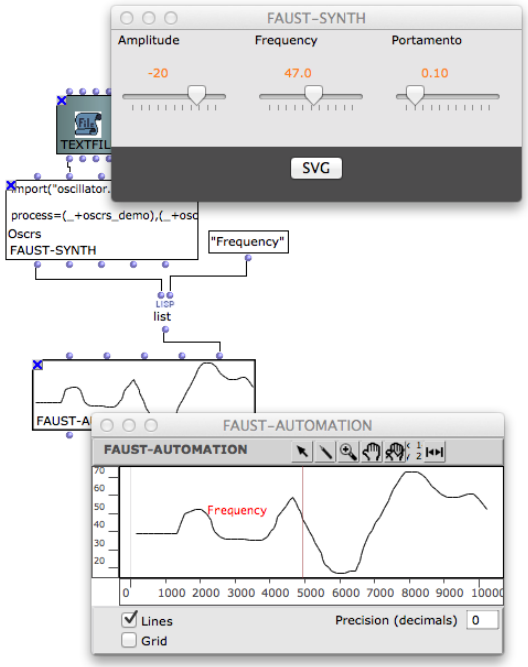


Figure 5.5 – Exemple d'objet faust-automation. Sa restitution contrôle un objet faust-synth en temps-réel.

une utilisation temps-réel de ces objets lorsqu'ils sont connectés à un flux audio géré par le système.

contrôle temps-différé / audio temps-réel. Il est possible d'entremêler composition temps-différé et génération temps-réel, notamment grâce à l'objet faust-automation (voir figure 5.5). Celui-ci permet de composer en temps-différé le contrôle des interfaces qui sera ensuite déroulé dans le temps au cours de la performance. Ainsi, la restitution d'une faust-automation est confiée à OpenMusic. La restitution d'une faust-automation produit des événements de contrôle qui sont pris en compte en temps-réel (aux frontières de buffer) sur le traitement des flux audio. Par conséquent, il est possible de composer le contrôle en temps-différé tout en le modifiant au cours de sa restitution.

L'approche temporelle mixte proposée par OM-Faust permet également l'intégration des objets Faust dans la maquette (voir section 1.2). En effet, l'évaluation de la maquette produit une structure statique (voir section 1.2.3) qui sera ensuite lue par un processus de restitution. Dans ce contexte, les objets Faust ne sont pas évalués en tant que fichiers sons (comme ils pourraient l'être grâce à flatten-faust-fx et flatten-

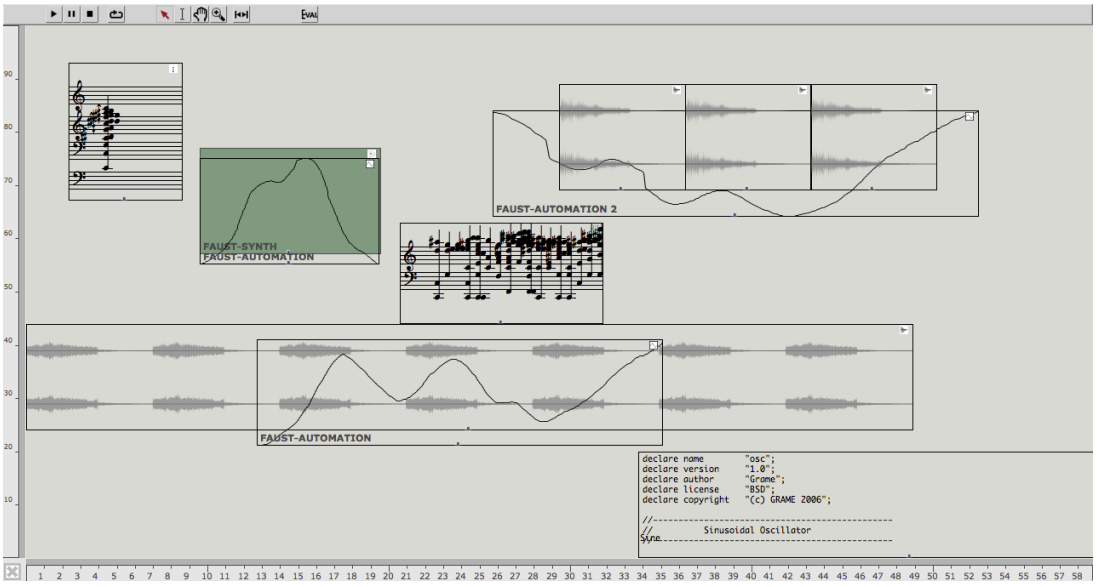


Figure 5.6 – Exemple de maquette (partition) intégrant des processus Faust.

faust-synth), mais sous la forme d'actions de déclenchement et d'arrêt, et leur contrôle est opéré par un utilisateur en temps-réel ou par une faust-automation. La figure 5.5 montre un exemple d'une telle maquette.

Ainsi, la restitution d'une partition contenant des objets OM-Faust consiste à faire opérer conjointement le moteur de restitution des actions d'OpenMusic et le moteur de restitution audio de LibAudioStream, l'un opérant sur des structures statiques/temps-différé, et l'autre générant des données en temps-réel. Ce modèle constitue un travail préliminaire d'une architecture audio plus complète que nous détaillons en section 5.3.

5.3 Une architecture audio asynchrone pour les processus compositionnels interactifs

Dans cette section, nous décrivons l'implémentation du moteur de restitution audio intégré à notre architecture d'ordonnancement.

5.3.1 *Objet son « temps-réel »*

Nous utilisons la fonctionnalité *real-time sound* de LAS, qui permet de spécifier une zone mémoire externe à la bibliothèque restituée comme un flux audio. Cela permet d'alimenter ou de modifier cette zone mémoire en cours de restitution.

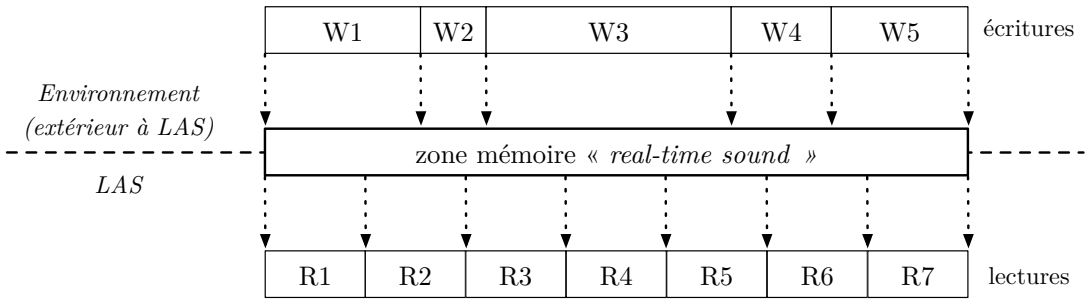


Figure 5.7 – Illustration des opérations d'écriture et de lecture de l'objet *real-time sound*.

Cet objet est une bonne illustration du modèle GALS (voir section 5.2.1) : tandis que le moteur interne à LibAudioStream vient copier des fragments réguliers de la mémoire vers la carte son de manière synchrone et périodique, l'environnement extérieur peut y écrire de manière asynchrone et irrégulière. Toute modification sur ce buffer affecte la restitution avec une latence maximale de b/SR ms où b est la taille de buffer interne au moteur audio de LAS, et SR sa fréquence d'échantillonnage. La figure 5.7 représente une illustration de ce principe. Nous décrivons ensuite comment nous avons implémenté une architecture audio autour de cet objet.

5.3.2 ..... *Implémentation de l'architecture audio*

Nous définissons la classe s-buffer dont tout objet devant manipuler des échantillons audio devra intégrer une instance (comme attribut), en conjonction avec l'héritage de classe s-object introduite précédemment (afin de pouvoir être géré par notre système d'ordonnancement). La classe s-buffer pointe vers une zone mémoire, elle-même liée à un *real-time sound* de LAS, et fournit une interface vers ses fonctions de transport (play, pause, continue, stop, jump-to).

Ainsi, tout objet de type s-object possédant un s-buffer comme attribut devra surcharger ses méthodes de transport héritées de la classe s-object afin d'y intégrer le contrôle du *real-time sound* (voir annexe A.2).

Nous introduisons donc un quatrième processus dans le système d'ordonnancement, chargé de la restitution des données audio (voir annexe B).

Prenons l'exemple de l'objet sound d'OpenMusic dont le rôle est simplement de charger et restituer des fichiers audio dans l'environnement. Son implémentation dans l'architecture que nous concevons se résume au transfert des échantillons d'un fichier audio présent sur le disque dur vers la zone mémoire allouée par l'objet s-buffer, puis de surcharger les méthodes de transport de la classe s-object. La redéfinition des méthodes de planification d'actions et de tâches (collect-actions et collect-tasks) de la classe s-

object est ici inutile. En effet, cet objet n'a aucune action ni tâche à exécuter : les échantillons sont une donnée statique, et la restitution est confiée à LAS. L'objet *real-time sound* permet cependant de modifier ces échantillons pendant la restitution. Nous en donnons des exemples en section 5.4.

Pour résumer, l'implémentation d'un objet sound dans notre système consiste à :

- copier les échantillons du fichier audio vers le s-buffer ;
- redéfinir les méthodes de transport, pour qu'elles redirigent vers les méthode de transport de LAS appliquées au s-buffer (les méthodes d'ordonnancement collect-actions et collect-tasks ne font rien).

Nous donnons son diagramme de séquence en annexe B.3.

5.4 Objets sonores dynamiques

Dans cette section, nous montrons comment le moteur de restitution audio présenté dans la section précédente permet l'intégration d'objets sonores dont les échantillons audio sont modifiés ou générés dynamiquement. Ces objets permettent de lier la composition – à un niveau structurel – à la génération et au contrôle temps-réel du signal audio.

5.4.1 *Requêtes audio apériodiques : IAE*

Dans cette section, nous donnons l'exemple d'un objet sonore dont l'écriture d'échantillons dans le s-buffer est apériodique.

IAE. ISMM Audio Engine (Cahen, 2012) est un moteur audio développé par l'équipe Interaction Son-Musique-Mouvement de l'Ircam. Sous forme d'une bibliothèque C++ portable, ce moteur a déjà été intégré dans Max, Unity3D, ou encore iOS. Le fonctionnement d'IAE repose principalement sur la synthèse granulaire et la synthèse concaténative par corpus basée sur du matériel audio annoté par des descripteurs (Schwarz, 2006). Les annotations sont faites sur des fragments de fichiers ou buffers audio de tailles variables et contiennent une description des caractéristiques de chaque segment : brillance, énergie, timbre, contenu fréquentiel, *etc.* L'annotation peut être produite automatiquement, extraite de fichiers créés par un autre logiciel, ou tout simplement spécifiée par l'utilisateur.

En pratique, l'API de la bibliothèque IAE permet d'envoyer des requêtes afin de générer des grains sonores à partir de fichiers audio sources. Nous avons créé un objet iae permettant le contrôle et l'organisation de ces requêtes dans le temps. Réalisé grâce

à la structure s-object vue en section 4.1.1, chaque élément (timed-item) représente une requête au moteur à une date donnée. Les paramètres de cette requête seront par exemple des positions et durées de grains dans un ficher source. Cette structure permet une utilisation en temps-différé d'IAE : il est possible d'organiser des requêtes dans le temps afin de les exécuter et de mixer les grains sonores produits dans une zone mémoire.

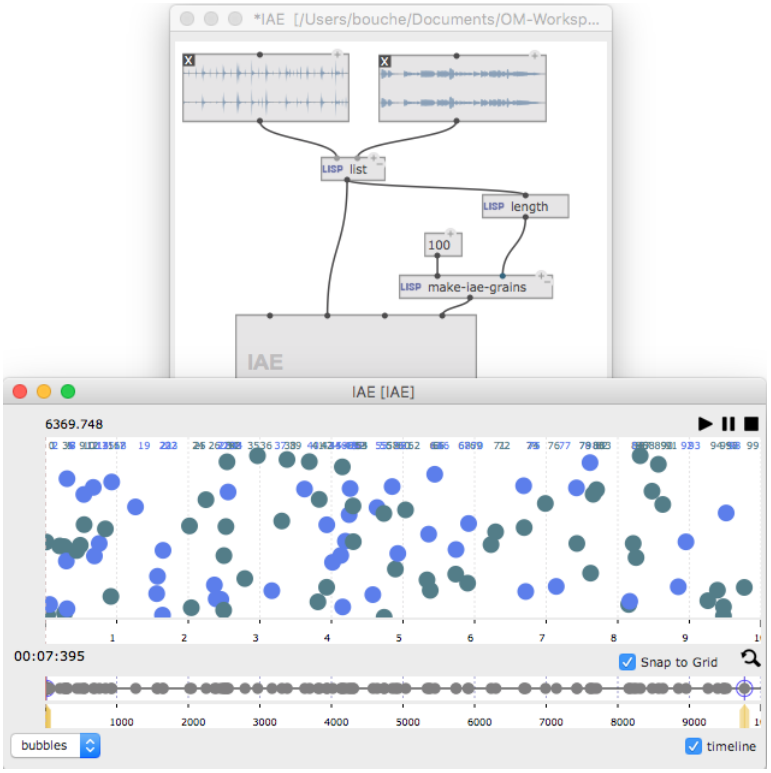


Figure 5.8 – Exemple de patch de création d'un objet IAE dans OM, avec 100 requêtes (aléatoires) sur deux fichiers sources. Chaque cercle correspond à une requête et un grain sonore sur la sortie audio temps-réel, sa couleur représente la source correspondante.

Une utilisation dynamique d'iae est prévue grâce à l'architecture audio présentée en section 5.3.2, couplée au système d'ordonnancement du chapitre 3. iae intègre un s-buffer correspondant à la zone mémoire restituée, qu'il peut alimenter en temps-réel et de manière asynchrone. Cette zone mémoire est initialement vide [6], et chaque requête envoyée à IAE afin de l'alimenter contient les opérations suivantes :

1. appel à IAE avec les paramètres de la requête ;
2. « synthèse » du grain correspondant par IAE ;

6. En pratique, elle est initialisée à 0 afin d'éviter tout problème de restitution audio.

3. écriture du grain dans la zone mémoire idoine du s-buffer.

L'objet iae informe le système d'ordonnancement des tâches à effectuer en surchargeant la méthode collect-tasks (voir section 4.2), et contrôle en parallèle la restitution de la zone mémoire du s-buffer en surchargeant les méthodes de transport de la classe s-object.

L'implémentation des méthodes d'un objet iae est la suivante :

- la méthode collect-actions ne fait rien ;
- la méthode collect-tasks renvoie une liste des requêtes datées dans l'intervalle d'ordonnancement ;
- les méthodes de transport redirigent vers les méthodes de transport de LAS appliquées au sound-buffer.

Pour résumer, l'objet iae permet donc :

- de contrôler la synthèse granulaire à l'aide d'une représentation structurée ;
- d'utiliser notre système d'ordonnancement afin d'éditer cette structure dynamiquement ;
- de déléguer la restitution au moteur temps-réel de LAS, tout en contrôlant l'écriture de manière asynchrone.

Nous donnons son diagramme de séquence en annexe B.4. Un exemple de patch permettant la création d'un objet IAE, ainsi qu'une visualisation de l'éditeur de ce même objet sont disponibles en figure 5.8.

5.4.2 ..... *Requêtes audio périodiques : Spat-scene*

Dans cette section, nous donnons l'exemple d'un objet sonore dont l'écriture d'échantillons dans un s-buffer est périodique.

spat. Le *spatialisateur* (Jot, 1999) est un moteur audio développé à l'Ircam dédié à la spatialisation sonore. Il se présente comme une bibliothèque C++/objective-C de composants audio et d'interfaces graphiques. Le moteur Spat est composé de deux modules : un réverbérateur artificiel multicanal (Jot et Chaigne, 1991) et un étage de panoramisation configurable (*panning* d'intensité ou de temps, encodage/décodage ambisonique, reproduction binaurale ou transaurale *etc.*). Il peut être contrôlé via une interface de contrôle perceptif (Jullien, 1995) pour créer un espace acoustique virtuel. Déployée dans divers environnements dédiés à la production musicale (objets Max, plugins VST/AU/AAX *etc.*), cette bibliothèque fait partie de l'extension OM-Spat permettant de spatialiser des fichiers audio en temps différé dans OpenMusic (Bresson, 2012).

L'objet spat-scene, développé par Garcia et al. (2016), fournit une interface utilisateur graphique pour définir les trajectoires sonores de sources sonores, en tirant avantage des capacités temps-réel de notre environnement de meta-CAO. Son éditeur graphique (voir figure 5.9) reprend l'interface de contrôle perceptif citée précédemment, en y associant une *timeline* pour chaque source audio (grâce à l'héritage de la classe s-object). L'utilisateur peut donc y disposer des points associant des dates à des positions dans l'espace, et ainsi faire se déplacer les sources dans l'espace acoustique.

La communication entre OpenMusic et le spatialisateur pour le calcul de sons spatialisés est réalisée via un système de requêtes à la bibliothèque, où chacune d'elle contient des informations spatiales (trajectoires) et audio (échantillons des fichiers sources).

Le mécanisme de restitution est similaire à celui de l'objet iae décrit précédemment, c'est à dire qu'il intègre un s-buffer et contrôle l'écriture dans sa zone mémoire dans le temps grâce aux requêtes collectées par l'ordonnanceur via la méthode collect-tasks. Chaque appel à collect-tasks renvoie une unique requête contenant les opérations suivantes :

1. appel au spatialisateur avec les portions des fichiers audio sources et les portions de trajectoires correspondantes ;
2. écriture des échantillons produits dans la zone mémoire idoine du s-buffer.

Il est possible d'utiliser l'objet spat-scene en temps-différé en utilisant une unique requête par objet, afin que spatialisateur produise un fichier multicanal. Une utilisation dynamique — permettant la prise en compte des modifications de trajectoire en cours de restitution — est aussi possible en démultipliant les requêtes et en réduisant la durée des paramètres passés en appel. Le résultat étant un signal continu, il est préférable dans ce cas de mettre en place des requêtes périodiques. Pour adapter le comportement de notre système à cette caractéristique, l'horizon d'ordonnancement (voir équation 3.4, section 3.2) est fixé à une certaine durée. Ainsi, la méthode collect-tasks sera appelée de manière périodique et ce pour des intervalles d'ordonnancement de longueurs égales, ce qui permet d'optimiser le processus de synthèse tout en préservant un comportement dynamique avec au plus un retard de l'ordre de l'horizon choisi.

Pour résumer, l'implémentation des méthodes d'un objet spat-scene est la suivante :

- la méthode collect-actions ne fait rien ;
- la méthode collect-tasks renvoie une tâche de synthèse dont la date est la borne inférieure de l'intervalle d'ordonnancement, et les paramètres sont les portions de trajectoires et d'échantillons audio de l'intervalle d'ordonnancement ;
- les méthodes de transport redirigent vers les méthodes de transport de LAS appliquées au sound-buffer.

Nous donnons son diagramme de séquence en annexe B.5.

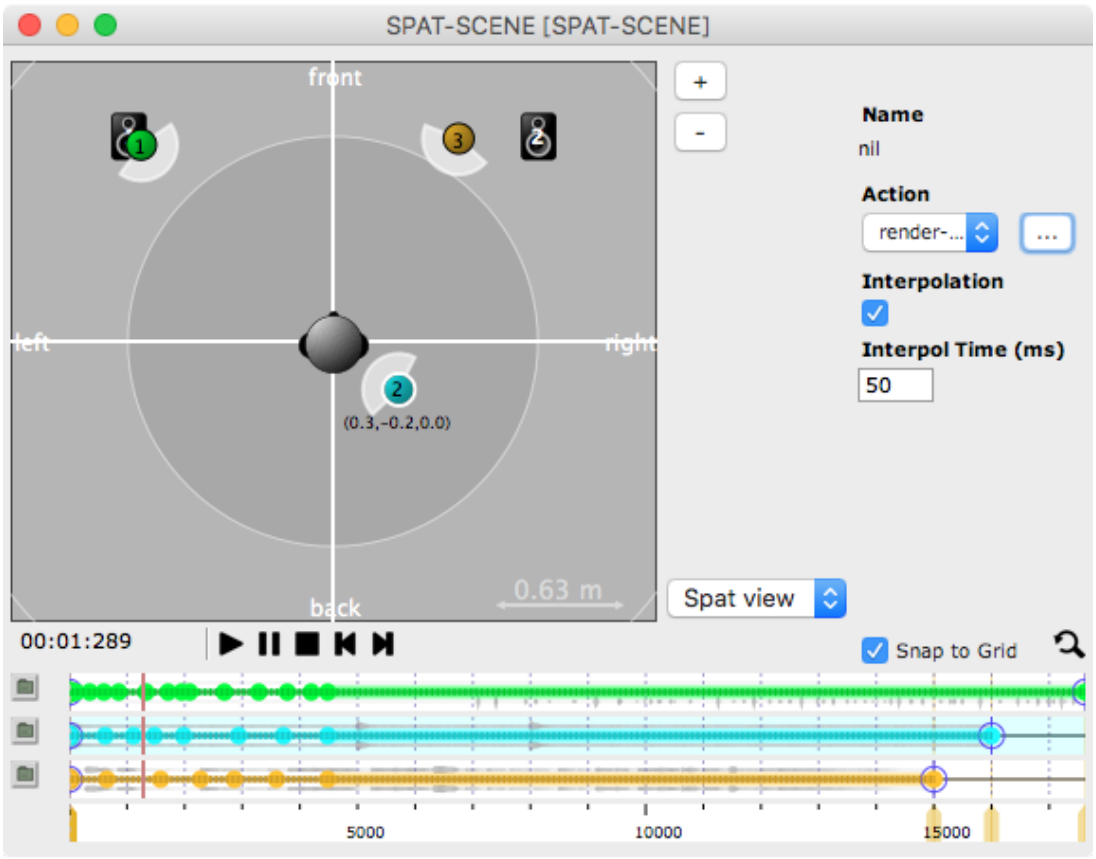


Figure 5.9 – Capture d'écran de l'éditeur d'un objet spat-scene permettant de spatialiser trois sources audio.

VI

FORMALISATION ET IMPLÉMENTATION DES META-PARTITIONS

Nous présentons dans ce chapitre un nouvel objet intégré à l'environnement de CAO OpenMusic permettant l'implémentation de processus de meta-composition. Cet objet est une extension de la maquette (présentée en section 1.2.3), appelée *meta-partition*. Sa structure dérive de la notion d'objet décrite en chapitre 4 et reliée au système d'ordonnancement décrit au chapitre 3. La meta-partition permet l'organisation temporelle de processus compositionnels couplée à des moyens d'interaction qui lui permettent d'opérer selon les trois modèles de calcul *demand-driven*, *event-driven* et *time-driven*. Nous en donnons une sémantique en section 6.2. Nous montrons son utilisation pour entremêler calcul et restitution en section 6.3, puis détaillons en sections 6.4 et 6.5 son interface et comment programmation et meta-programmation peuvent s'articuler. Ce chapitre se termine en section 6.6 avec des exemples de meta-compositions.

6.1 Sémantique et formalisme

Une meta-partition correspond à un programme visuel (patch), déroulé dans le temps, dont la restitution peut modifier le patch lui-même. Nous donnons en section 6.1.1 une sémantique formelle des programmes visuels dans OpenMusic, inspirée de Bresson et Giavitto (2014), que nous étendons en section 6.1.2 à une sémantique des maquettes qui nous permettra par la suite de donner une sémantique de la meta-partition.

6.1.1 *Sémantique des patches*

L'article de Bresson et Giavitto (2014) donne une sémantique des programmes visuels dans OpenMusic et de leur extension réactive. Les définitions qui suivent sont adaptées de celles données dans cet article, en intégrant les plans comme valeurs des boîtes manipulées.

STRUCTURE. Soit $\mathcal{B}$ l'ensemble des identifiants des boîtes (ensemble des nœuds possibles dans un patch) et $\mathcal{E}$ l'ensemble d'identifiants des arêtes d'un patch. Les éléments de $\mathcal{B}$ seront notés $b, b', b_1 \ldots$ et $e, e', e_1 \ldots$ pour $\mathcal{E}$.

Les fonction *in* et *out* sont des fonctions totales de $\mathcal{B}$ vers $\mathbb{N}$, renvoyant respectivement le nombre d'entrées et de sorties d'une boîte. Un patch est un quadruplet

$$G = (\mathsf{B}, \mathsf{E}, \mathsf{s}, \mathsf{t}), \tag{6.1}$$

avec $\mathsf{B} \subset \mathcal{B}$ l'ensemble fini de boîtes de G, $\mathsf{E} \subset \mathcal{E}$ l'ensemble fini des arêtes entre ces boîtes, et s (origine d'une arête) et t (cible d'une arête) sont des fonctions totales de E vers $\mathsf{B} \times \mathbb{N}$ représentant la connectivité dans un patch : si l'arête e relie la $j^{\text{ème}}$ sortie de la boite b' avec la $i^{\text{ème}}$ entrée de la boîte b, alors $s(e) = (b', j)$ et $t(e) = (b, i)$.

ÉVALUATION. L'*évaluation* d'un patch consiste en l'affectation d'une valeur à chacune de ses boîtes. Pour notre système, la valeur d'une boîte est sa trace équivalente.

Pour chaque boîte $b \in \mathcal{B}$, il y a $out(b)$ fonctions associées

$$[\![b]\!]_k : \mathcal{T}^{in(b)} \to \mathcal{T}, \quad 1 \leq k \leq out(b) \tag{6.2}$$

fournissant la sémantique d'une boîte.

EXTENSION SYNCHRONE ET ASYNCRHONE. Une boîte OpenMusic classique avec une seule sortie correspond à une fonction (implémentée en Lisp) et il y a plusieurs manières de l'étendre en une fonction qui transforme des plans d'actions. L'*extension synchrone* consiste à ne combiner que des valeurs en entrée qui sont présentes simultanément : soit f la fonction associée à une boîte $b : \mathcal{T}^{in(b)} \to \mathcal{T}$. Alors l'extension synchrone de f est la fonction $F : \mathcal{T}^{in(b)} \to \mathcal{T}$ définie par $F(p_1, ..., p_{in(b)}) = p$ avec p défini par $(t, v) \in p \Leftrightarrow \exists v_1, \ldots, v_{in(b)}$ tel que $v = f(v_1, \ldots, v_{in(b)})$ et $(t, v_i) \in p_i$ pour $1 \leq i \leq in(b)$. Autrement dit, une valeur est produite en sortie à la date t quand toutes les entrées sont présentes simultanément à la date t.

Cependant, cette extension n'est pas pertinente dans le cadre réactif d'OpenMusic. En effet, si un événement met à jour une seule entrée de boîte, on veut recalculer la valeur de sortie en prenant pour valeurs des autres entrées leur ancienne valeur. Cette stratégie correspond à l'*extension asynchrone*. L'extension asynchrone de f est la fonction $F : \mathcal{T}^{in(b)} \to \mathcal{T}$ définie par $F(p_1, ..., p_{in(b)}) = p$ avec p défini par $(t, v) \in p \Leftrightarrow \exists (t_1, v_1), \ldots, (t_{in(b)}, v_{in(b)})$ tel que $v = f(v_1, \ldots, v_{in(b)})$ et $(t_i, v_i) \in p_i$ et $t_i = last_t(p_i{\downarrow}^t)$ pour $1 \leq i \leq in(b)$ et $\exists j \in \{1, \ldots, in(b)\}, (t, a_j) \in p_j$.

Ces fonctions sont admises continues dans le cadre de la théorie des domaines (van Leeuwen, 1990).

SÉMANTIQUE D'UN PATCH. La sémantique d'un patch $G = (\mathsf{B}, \mathsf{E}, \mathsf{s}, \mathsf{t})$ est une fonction $[\![.]\!]_G(.) : \mathsf{B} \times \mathbb{N} \to \mathcal{T}$ associant un plan à chaque sortie k de boîte $b \in \mathsf{B}$ et définie par l'équation récursive

$$[\![b]\!]_G(k) = [\![b]\!]_k(v_1, \ldots, v_{in(b)}), \tag{6.3}$$

où

$$v_i = [\![b']\!]_G(j) \quad \text{si } b \ {}_i{<}_j \ b' \tag{6.4}$$

où $b \ {}_i{<}_j \ b'$ signifie que la sortie j d'une boîte b' est connectée à l'entrée i d'une boîte b. On voit ici que les plans sont utilisés dans un calcul compositionnel, comme énoncé en section 3.1.3.

6.1.2 *Sémantique des maquettes*

STRUCTURE. Nous pouvons étendre le formalisme précédent en définissant une maquette comme le quadruplet

$$M = (\mathsf{B}, \mathsf{X}, \mathsf{E}, \mathsf{s}, \mathsf{t}), \tag{6.5}$$

avec X une fonction qui associe à chaque boîte $b \in \mathsf{B}$ une position $(x, y) \in \mathbb{R}^2$ et $\mathsf{B}, \mathsf{E}, \mathsf{s}, \mathsf{t}$ définis comme en équation 6.1. La dimension horizontale d'une maquette représente le temps, donc en réalité $\mathsf{X}(b) \subset (\mathcal{D} \times \mathbb{R})^*$ (en pratique, $\mathcal{D} = \mathbb{N}$).

ÉVALUATION. L'évaluation d'une maquette est identique à celle d'un patch et consiste en l'affectation d'une valeur à chacune de ses boîtes. A la différence d'un patch, les coordonnées d'une boîte peuvent être utilisées comme paramètre de sa propre évaluation, et son évaluation peut agir sur ces mêmes coordonnées. Ainsi, l'évaluation de la maquette M produit, en plus du plan d'action, une nouvelle maquette M^e qui correspond à l'évolution de la maquette elle-même suite à son évaluation. La nouvelle maquette M^e est spécifiée par

$$M^{\mathsf{e}} = (\mathsf{B}, \mathsf{X}^{\mathsf{e}}, \mathsf{E}, \mathsf{s}, \mathsf{t}) \tag{6.6}$$

qui représente le graphe post-évaluation. On remarque que le graphe sous-jacent défini par $(\mathsf{B}, \mathsf{E}, \mathsf{s}, \mathsf{t})$ n'est pas modifié, mais que les positions spatiales de ses boîtes peuvent changer.

Le graphe M^{e} permet de créer un plan de restitution à partir de la valeur des boîtes qu'il contient : chaque valeur de boîte est une trace $\mathcal{T}$, et la valeur de M^{e} est l'agrégation de ces traces.

6.2 Structure et évaluation d'une meta-partition

Dans cette section, nous formalisons un nouvel objet OpenMusic permettant la meta-CAO : la *meta-partition*. Une meta-partition est un programme visuel au sens de la définition 6.5, dont les éléments sont datés et peuvent être des abstractions (processus compositionnels) non calculées avant le démarrage de la restitution.

structure. La meta-partition est une extension de la maquette (section 6.1.2) qui peut être modifiée au cours du temps. Par conséquent, la modélisation de ses états sous forme d'un couple pré/post-évaluation similaire à une maquette n'est plus valide : une meta-partition peut posséder un nombre d'états successifs inconnus à l'avance. Ces états peuvent se distinguer par des modifications du graphe équivalent, et non seulement des positions spatiales des boîtes comme c'était le cas pour la maquette. Les états de la meta-partition peuvent donc se formaliser sous la forme d'une suite de quintuplets

$$M^t = (\mathsf{B}^t, \mathsf{X}^t, \mathsf{E}^t, \mathsf{s}^t, \mathsf{t}^t) \tag{6.7}$$

où les éléments du quintuplets sont identiques à la définition 6.5, et où M^t est le $t^{\text{ème}}$ état de la partition. On notera $\mathcal{M}$ l'ensemble des meta-partitions.

Les boîtes contenues dans une meta-partition sont des abstractions (voir section 1.2.1) et des boîtes « isolées ». Une abstraction a pour plan équivalent une action a_c déclenchant une tâche. Les boîtes isolées sont des boîtes possédant une valeur fixée au cours du temps (par exemple une séquence de notes ou un fichier audio) : leur plan est celui correspondant à cette valeur.

abstractions. Nous donnons ici une sémantique des abstractions comme définies en section 1.2.1. Notons $b^o_1, \ldots, b^o_{n_o}$ les n_o boîtes *output* d'une abstraction. Nous rappelons qu'une boîte *output* b^o ne possède qu'une entrée, et a donc pour sémantique d'évaluation $[\![b^o]\!] = [\![b^o]\!]_1(v_1)$ avec $v_1 = [\![b]\!]_G(k)$ si $b^o <_k b$ (la sortie k d'une boîte b est connectée à b^o). Le calcul d'une abstraction est l'évaluation des n_o boîtes *output* qu'elle contient : $[\![b^o]\!]_G \Rightarrow [\![b^o_1]\!] \ldots [\![b^o_{n_o}]\!]$. Par convention, étant donné qu'une abstraction peut posséder plusieurs sorties, sa valeur sera celle de la première sortie $[\![b^o_1]\!]$.

évaluation. Tout comme une maquette, la meta-partition est définie comme un programme visuel. Cependant, son évaluation dépend du temps : les interactions utilisateurs, événements et calculs déclenchés par sa propre restitution induisent des ré-évaluations. La sémantique d'évaluation d'une meta-partition $M = (\mathsf{B}, \mathsf{X}, \mathsf{E}, \mathsf{s}, \mathsf{t})$ est donc une fonction $[\![.]\!] : \mathcal{M} \to \mathcal{T}$ associant une trace à une meta-partition obtenue à partir de la trace de chaque boîte b qu'elle contient.

$$[\![M]\!] = \sum_{b \in \mathsf{B}} \sum_{k=1}^{out(b)} [\![b]\!]_k(v_1, \ldots, v_{in(b)}) \tag{6.8}$$

où v_i est défini comme à l'équation 6.4.

exemple. Prenons l'exemple d'une meta-partition M (à l'état initial M^0) contenant (uniquement) une boîte d'abstraction b, placée à la date d_0. L'évaluation de la meta-partition produit un plan singleton : $[\![M]\!] = (d_0, a_c)$. Si l'on note $p_M(t)$ le

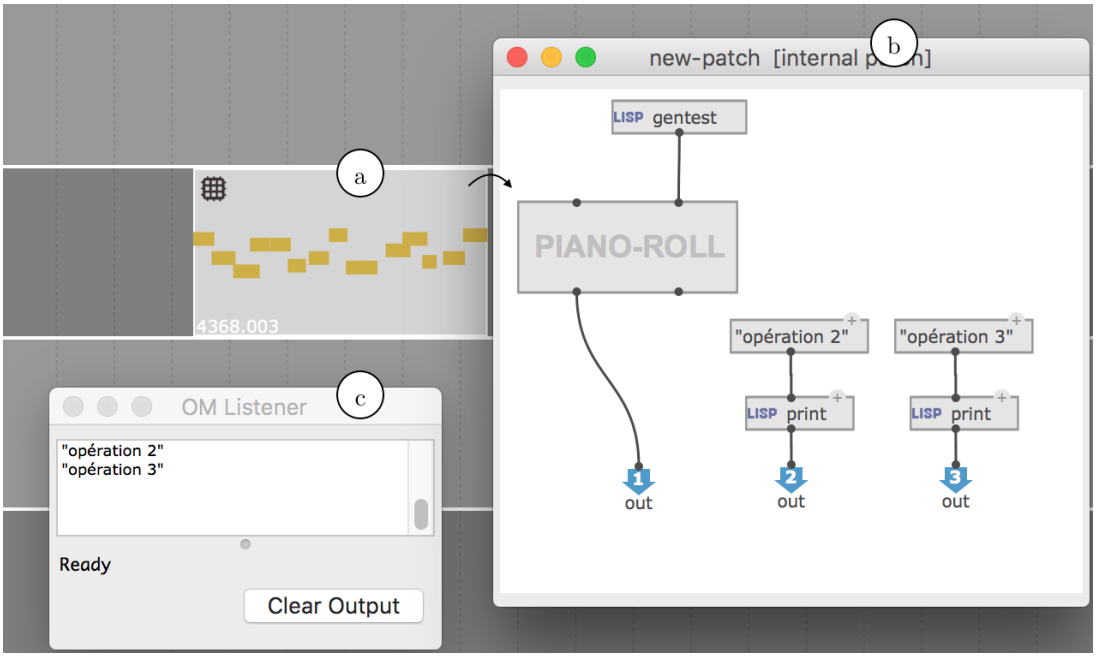


FIGURE 6.1 – Évaluation d'un abstraction contenant trois boîtes *output* dans une meta-partition : sa valeur est celle de sa première boîte *output* (ici un piano-roll), mais elle produit également les valeurs des autres boîtes (ici, des affichages).

plan de la meta-partition M, on a selon les définitions vues en section 3.1.4 : si $p_M(t)\uparrow_t = (d_0, a_c)$ alors $p_M(t) \neq p_M(t_0)$, car à $t_0 = eval(d_0)$ l'action a_c modifie le plan de M lors de l'évaluation de b. L'action a_c modifie le plan de M, et induit donc son passage de l'état M^0 à $M^1 = a_c(M^0)$.

REPRÉSENTATION GRAPHIQUE. Lorsqu'une abstraction est évaluée dans une meta-partition, sa représentation graphique est mise à jour afin de refléter la valeur de sa première boîte d'*output*. Par exemple, si sa valeur est une séquence de notes, la représentation graphique de la boîte est un piano-roll. La figure 6.1 résume cette situation : une abstraction (a) est placée dans l'interface que nous concevons et est évaluée. Cette abstraction correspond au patch (b), qui possède trois sorties. L'évaluation de (a) provoque l'évaluation des trois sorties du patch. L'élément (a) prend alors la valeur de l'évaluation de sa première sortie (un piano-roll), mais les deux autres sorties sont quand même évaluées : on peut le voir dans le *listener* (c) qui affiche les résultats de toutes les évaluations du programme. Cela permet non seulement aux processus compositionnels de produire des structures musicales, mais également de déclencher des opérations arbitraires. Ces opérations peuvent être des messages, des modifications des éléments de la meta-partition, de l'affichage *etc.*

6.3 Lien au moteur d'ordonnancement

La meta-partition est un objet au sens de la définition vue en section 3.1.1, implémenté grâce à la structure s-object vue en chapitre 4. Sa restitution peut collecter récursivement les actions des objets qu'elle contient, ou déclencher leur ordonnancement parallèle (restitution hiérarchique/simple, voir section 4.2.2).

L'implémentation de la méthode collect-actions suit actuellement un fonctionnement « hybride » (voir algorithme 3) : hiérarchique dans le cas où l'objet à restituer contient un s-buffer (auquel cas seul le déclenchement de la restitution sonore est nécessaire, ensuite gérée par le système audio — voir chapitre 5), et simple dans les autres cas.

Algorithm 3 Collection des actions dans une meta-partition

$\mathcal{B} \leftarrow$ ensemble de boîtes dans la partition
$d_b \leftarrow$ date d'une boîte b
$l_b \leftarrow$ durée d'une boîte b
$\mathcal{I}$ intervalle d'ordonnancement
$p \leftarrow null$ plan produit

for $b \in \mathcal{B}$ **do**
  if $(\exists$ s-buffer $\in b) \wedge (eval(d_b) \in \mathcal{I})$ **then**
    ▷ La boîte utilise un s-buffer
    $p \leftarrow p +_p (eval(d_b), \mathsf{play}(b))$
  else if $(b \sim$ s-object$) \wedge ([eval(d_b); eval(d_b + l_b)[\cap \mathcal{I} \neq \emptyset)$ **then**
    ▷ La boîte a une valeur d'objet
    $p \leftarrow p +_p$ collect-actions$(b, \mathcal{I})$
  end if
end for
return p

Algorithm 4 Collection des tâches dans une meta-partition

for $b \in \mathcal{B}$ **do**
  if $dynamic(b) \wedge \ (eval(d_b) \in \mathcal{I})$ **then**
    ▷ La boîte est une abstraction réactive
    $p \leftarrow p +_p (eval(d_b), a_c(b))$
  end if
end for
return p

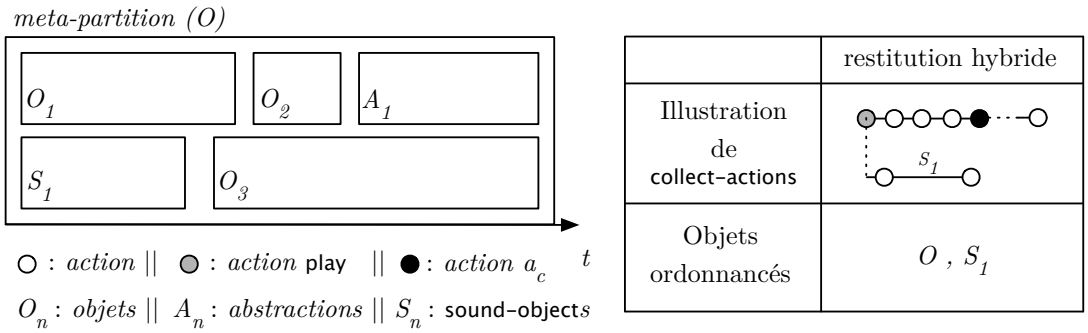
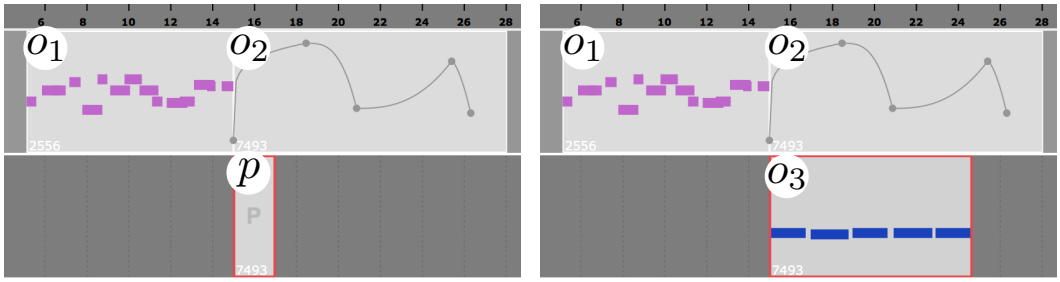


FIGURE 6.2 – Illustration de la restitution d'un meta-partiton.

Indifféremment du mécanisme préféré, toute abstraction dynamique est collectée par la méthode collect-tasks comme une action a_c (voir section 3.1.4), de date égale à l'abscisse de la boîte dans l'interface (voir algorithme 4). Les autres éléments considérés comme ayant une valeur fixe sont donc gérés par l'ordonnanceur comme l'ensemble d'actions atomiques équivalent à leurs valeurs. La figure 6.2 illustre ce fonctionnement.

La figure 6.3 montre la restitution d'une meta-partition contenant une abstraction dynamique ainsi que deux objets dont la valeur est déjà fixée (par exemple, composés manuellement par l'utilisateur).



(a) Avant la date de la boîte d'abstraction (b) Après la date de la boîte d'abstraction

FIGURE 6.3 – Restitution d'une meta-partition contenant une abstraction.

On peut observer que l'abstraction est calculée lorsque le temps de la restitution est égal à la date de la boîte. Selon le formalisme vu en section 3.2, on a [1]

$$\forall t < t_p,\ p_o(t) = p_{o_1}(t) + p_{o_2}(t) + (eval(d_p), a_c).$$

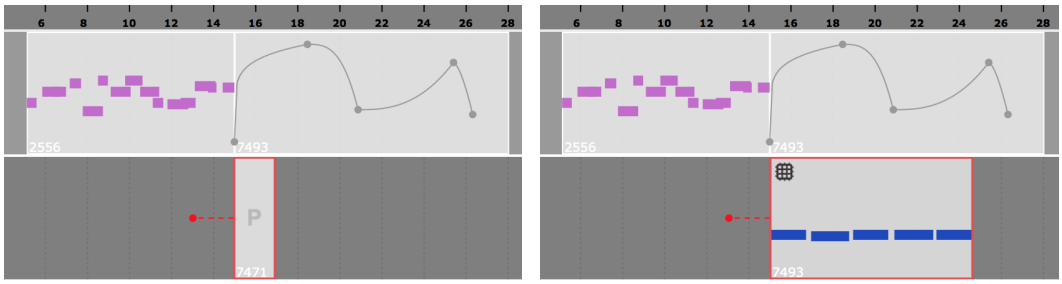
1. Par souci de simplification, nous considérons dans cette équation et dans la suite de ce manuscrit que l'opérateur *eval* ne dépend pas du temps, ce qui est possible en pratique.

où d^p est la date à laquelle l'abstraction est positionnée, et $t_p = eval(d_p)$. Après l'exécution de a_c au temps t_p, si l'on note $\Delta_{call(a_c)}$ la durée du calcul de l'abstraction, on a

$$p_o(t_p + \Delta_{call(a_c)}) = p_{o_1}(t) + p_{o_2}(t) + p_{o_3}(t).$$

anticipation. Dans un cas de figure simple comme celui présenté en figure 6.3 (génération d'une courte séquence de notes), $\Delta_{call(a_c)}$ est négligeable et le résultat du calcul est intégré sans latence notable. Cependant, dans le cas d'abstractions plus complexes dont le calcul est de durée non négligeable, le résultat peut être intégré en retard et les premières actions de son contenu manquées par la restitution. Dans ce cas, l'hypothèse synchrone n'est plus vérifiée.

Lorsque l'hypothèse synchrone n'est pas vérifiée, il est parfois possible d'anticiper le calcul afin de disposer de plus de temps pour le réaliser tout en connaissant la valeur exacte de ses paramètres. Cette anticipation est difficile à calculer automatiquement C'est pourquoi nous proposons un mécanisme permettant au compositeur de spécifier cette valeur explicitement en associant un *pre-delay* à un élément. Cette valeur détermine une date à partir de laquelle l'évaluation de la boîte peut être déclenchée. La figure 6.4 montre un tel scenario, où l'évaluation de la boîte d'abstraction est autorisée avec une seconde d'anticipation.



(a) Avant la date d'anticipation (b) Après la date d'anticipation

Figure 6.4 – Restitution d'une meta-partition contenant une boîte d'abstraction dynamique avec un *pre-delay* (anticipation autorisée).

Ce mécanisme ne modifie pas les sémantiques présentées : une anticipation revient seulement à la translation temporelle d'une action a_c dans le plan d'une meta-partition. Cependant, il introduit une durée entre la date à laquelle a_c est déclenchée, et la date (position) de l'abstraction dans la meta-partition, date maximale à laquelle un résultat doit être fournit. La conséquence principale du mécanisme d'anticipation est que les tâches arrivent dans le calculateur dans un ordre qui n'est plus celui souhaité en sortie. Le tri de la file d'attente du moteur de calcul de notre système permet d'optimiser la

complétion des tâches à temps, mais des algorithmes d'ordonnancement plus complexes peuvent être envisagés (voir annexe C).

Cela permet de compenser les temps de calcul, mais aussi de préciser une date à laquelle seront lues les valeurs des paramètres d'un processus. Par exemple, si le processus compositionnel en figure 6.4 construit la séquence de notes à partir d'une valeur évoluant constamment, une anticipation d'une seconde la lira une seconde avant le démarrage de la restitution de l'objet.

Si l'on note $\Delta_{pre_{a_c}}$ la valeur du pre-delay, le plan équivalent de la méta-partition avant déclenchement de a_c devient

$$\forall t < t_p,\ p_o(t) = p_{o_1}(t) + p_{o_2}(t) + (t_p - \Delta_{pre_{a_c}}, a_c).$$

Après exécution de a_c,

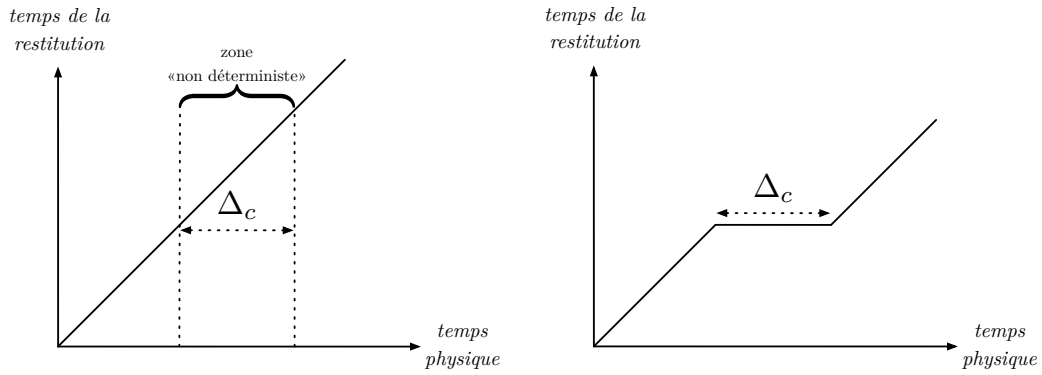
$$p_o(t_p - \Delta_{pre_{a_c}} + \Delta_{call(a_c)}) = p_{o_1}(t) + p_{o_2}(t) + p_{o_3}(t).$$

La condition pour que le résultat du calcul soit intégré à la partition sans latence notable est donc $\Delta_{call(a_c)} < \Delta_{pre_{a_c}} + \delta_{min}$. En effet, nous rappelons que, comme vu en section 3.2, la prise en compte d'une modification d'un objet dans la restitution se fait avec une latence fixe de δ_{min}, qu'il faut donc prendre en compte dans la mise en place de $\Delta_{pre_{a_c}}$.

RESTITUTION INTERRUPTIBLE. Le mécanisme d'anticipation décrit ci-dessus permet à un compositeur de compenser manuellement la complexité des calculs déclenchés par une meta-partition. Le système que nous proposons est optimisé pour ce genre de réalisations, mais ne fournit pas de garanties de terminaison des calculs en temps voulu. Il existe donc une forme d'indéterminisme dans le processus de restitution : si le résultat d'un calcul arrive en retard, la restitution peut manquer certaines de ses actions.

Afin de rendre la restitution d'une meta-partition déterministe, nous intégrons une option permettant d'interrompre la restitution à chaque calcul déclenché. Si un calcul est déclenché par la restitution, cela revient simplement à intégrer l'exécution des actions a_c dans la boucle de répartition plutôt que de les rediriger vers le calculateur. Dans le cas d'un calcul déclenché par une interaction, nous intégrons les appels aux méthodes pause et continue respectivement au démarrage et à la complétion du calcul.

Cette manière de procéder permet à l'utilisateur de réaliser de la composition contrôlée « incrémentale » : il pourra voir avec précision les conséquences de chaque interaction. Remarquons que si les exécutions sont suffisamment rapides, les discontinuités dans la restitution peuvent ne pas être perceptibles. La figure 6.5 donne une illustration de l'écoulement du temps dans ce mode de fonctionnement. Nous proposons une option pour que l'utilisateur puisse activer ce fonctionnement.



(a) Restitution non-interruptible : la zone temporelle définie par la durée d'un calcul est « non déterministe » : certains événements peuvent être manqués.

(b) Restitution interruptible : la restitution s'interrompt pendant le temps du calcul, et aucun événement n'est manqué.

Figure 6.5 – Illustration de l'écoulement du temps dans la restitution d'une meta-partition.

6.4 Interface

6.4.1 *Vue groupée*

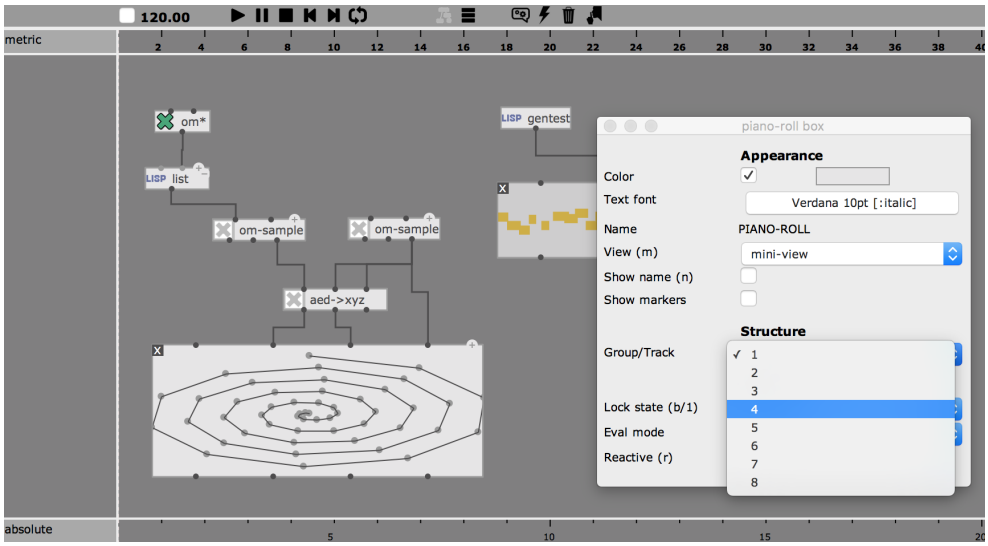
Certains composants de notre meta-partition ont pour seule tâche de calculer d'autres composants, et leur visualisation n'est pas forcément souhaitable durant la restitution. En ce sens, nous proposons une interface visuelle « duale » :

- d'un côté, une présentation du programme visuel identique au modèle de la maquette d'OpenMusic, à savoir un patch dont la dimension horizontale représente le temps, et la dimension verticale peut être utilisée comme paramètre de calcul ;
- de l'autre, une présentation sous formes de pistes superposées (à l'instar de la plupart des séquenceurs modernes) où seuls les objets sélectionnés par l'utilisateur apparaissent.

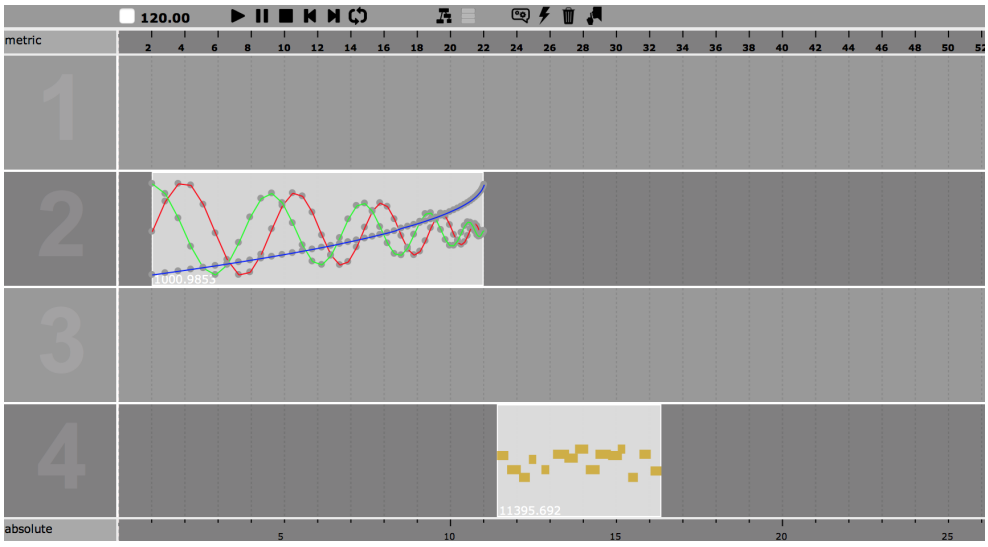
Pour cela, un numéro de groupe peut être assigné à chaque élément du programme visuel, correspondant à une piste. La figure 6.6a illustre l'affectation d'un groupe à un objet piano-roll.

6.4.2 *Règles temporelles*

Temps absolus et symboliques. La meta-partition présente deux *règles* complémentaires, permettant l'organisation de son contenu suivant des temps absolus ou des temps symboliques, fonctions d'un tempo. Chacune d'elle intègre une fonction



(a) Vue « maquette » : assignation d'un groupe pour une boîte.



(b) Vue « pistes » : organise les boîtes de la vue « maquette » selon leur groupe.

FIGURE 6.6 – Deux visualisation d'une même meta-partition : vue « maquette » et vue « pistes ». Le passage d'une vue à l'autre est possible en un *clic*.

snap to grid – quantification automatique de déplacement – permettant au compositeur de disposer avec précision les objets et abstractions suivant le temps symbolique et/ou absolu. Ces règles intègrent un mécanisme de collecte des timed-items *master* des objets contenus dans la meta-partition. Ces marqueurs sont représentés par des flèches jaunes, agrégées au sein des règles, et bénéficient de la correction automatique de déplacement. Leur affichage peut être activé ou non pour chaque boîte individuelle. Cette caractéristique renvoie à la notion de « remontée des pivots » de Stroppa et Duthen (1990).

SYNCHRONISATION. La collection de marqueurs de multiples objets d'une meta-partition permet de visualiser leurs points de synchronisation sur une règle unique et de les synchroniser.

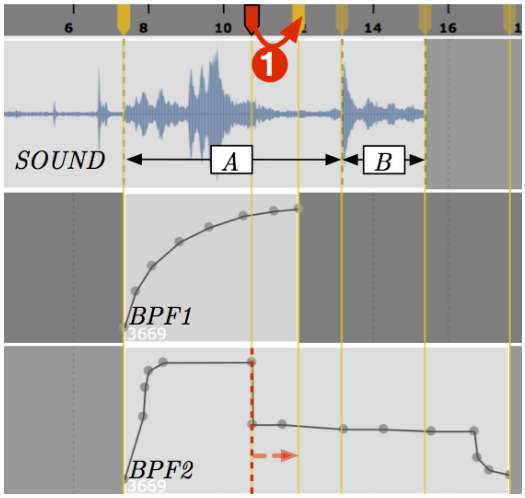
La figure 6.7 illustre une situation simple dans laquelle l'utilisateur décide de synchroniser deux objets bpf contrôlant des effets appliqués à un fichier audio. Le fichier audio est annoté avec deux marqueurs représentant deux sections A et B. Un élément dans la seconde bpf est défini comme *master* (on peut observer le marqueur correspondant en (1)), afin d'être synchronisé avec un des marqueurs du fichier audio.

En figure 6.7a, le premier élément de chaque objet bpf est synchronisé avec le premier marqueur du fichier audio (début de la section A). L'action utilisateur (1) synchronise ensuite l'élément *master* de $BPF2$ (marqueur en rouge sur la règle) avec le dernier élément de $BPF1$. Cette action lie les éléments ensemble. Ils peuvent être manipulés ensuite comme une seule entité.

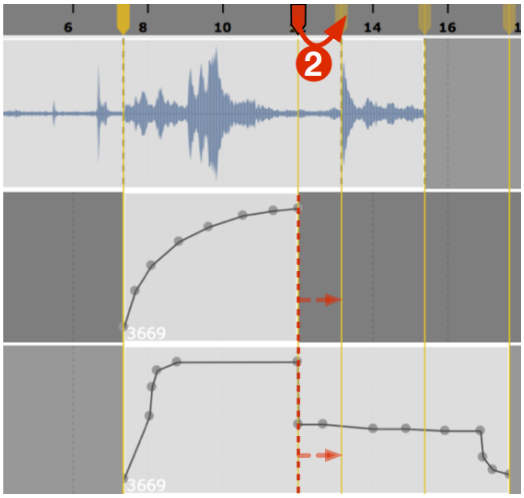
En figure 6.7b, l'utilisateur réalise l'action (2) afin de placer le marqueur sélectionné au début de la section B du fichier audio. Finalement, en figure 6.7c l'action (3) connecte le dernier élément de $BPF2$ à la fin du fichier audio, afin d'étirer temporellement le second segment de l'objet bpf. La 6.7d montre la position finale des objets après ces différentes opérations de synchronisation.

6.4.3 ..... *Outils*

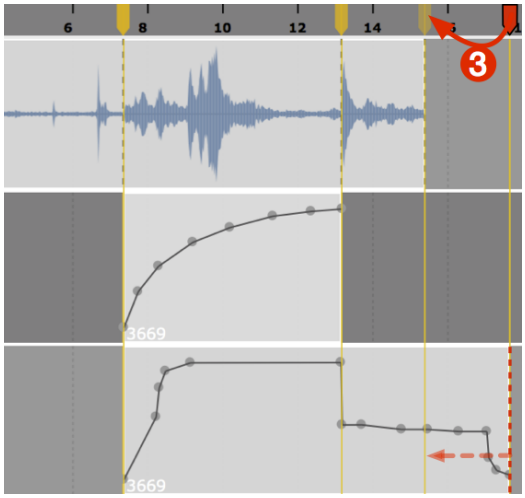
Certaines caractéristiques de la meta-partition décrites dans les sections précédentes peuvent être contrôlées par l'utilisateur. Tout d'abord, différents aspects des boîtes peuvent être manipulés grâce à un *inspecteur*, fenêtre unique affichant les paramètres d'une boîte sélectionnée. Par exemple, sur la figure 6.6a, on peut observer l'inspecteur d'un piano-roll permettant de choisir son groupe (piste), afficher ou masquer ses marqueurs de synchronisation *etc.* Selon le type de boîte sélectionnée, l'inspecteur proposera différentes options. Par exemple, il est possible d'y modifier le *pre-delay* d'une abstraction (voir section 6.3), ou encore préciser si elle est dynamique.



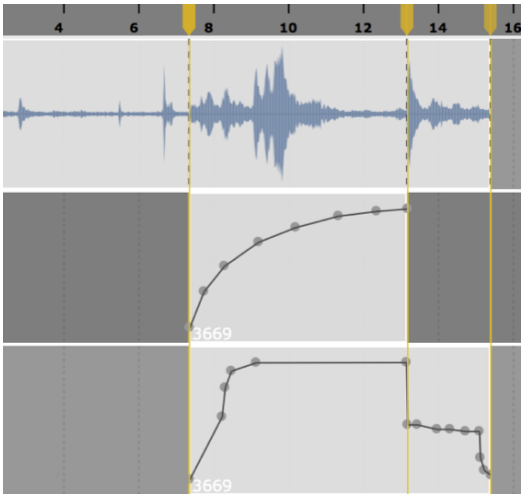
(a) Synchronisation d'un point de $BPF2$ avec la fin de $BPF1$.



(b) Synchronisation simultanée du « milieu » $BPF1$ et de la « fin » de $BPF2$ début de la section B de $SOUND$.



(c) Synchronisation de la fin de $BPF2$ avec la fin de $SOUND$.



(d) État final de la meta-partition.

Figure 6.7 – Synchronisation de deux bpf contrôlant des effets appliqués à un fichier audio (1, 2 et 3 sont des interactions utilisateur).

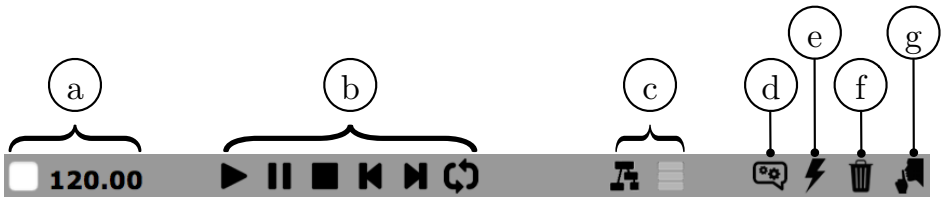


Figure 6.8 – Barre d'outils de la meta-partition.

La meta-partition présente également une barre d'outils (voir figure 6.8) permettant de contrôler son comportement ainsi que son affichage :

— (a) : activation/désactivation du tempo,
— (b) : méthodes de transport,
— (c) : choix du mode de présentation (vue maquette/pistes),
— (d) : affichage du patch de contrôle (voir section 6.5.2),
— (e) : évaluation totale de la meta-partition,
— (f) : suppression du contenu de la meta-partition,
— (g) : désactivation des actions musicales (m-with-exec, voir section 6.5.1).

6.5 Programmation et meta-programmation

6.5.1 *API de la meta-partition*

La restitution d'une meta-partition permet le déclenchement de calculs venant modifier son contenu. Pour cela, elle fournit un ensemble de trois opérateurs permettant d'ajouter, supprimer ou déplacer des éléments. Ces opérateurs peuvent être utilisés dans des programmes ou processus compositionnels Dans les définitions qui suivent, nous notons o un objet arbitraire et m la meta-partition qui le contient.

— m-add(m, o, d) : ajoute un objet o dans une meta-partition m à une date d ;
$\equiv p_m(t+1) \leftarrow p_m(t) + p_o(t)$
— m-remove(m, o) : supprime un objet o d'une meta-partition m ;
$\equiv p_m(t+1) \leftarrow p_m(t) - p_o(t)$
— m-move$(m, o, \Delta d)$: déplace un objet o dans une meta-partition m en ajoutant Δd à sa date.
$\equiv p_m(t+1) \leftarrow p_m(t) - p_o(t) + (eval(\Delta d), null).p_o(t)$

Nous proposons également un ensemble d'accesseurs et d'opérateurs de contrôle, relatifs au contenu et à l'état de la meta-partition à un instant donné :

— m-play(s) : démarre ou poursuit la restitution de s,

— m-pause(s) : interromp la restitution de s,
— m-stop(s) : arrête la restitution de s,
— m-loop(s,t_1,t_2) : boucle la restitution de s entre t_1 et t_2,
— m-get-time(s) : récupère la date courante de restitution de s,
— m-set-time(s,t) : change le temps de restitution de s pour la valeur t,
— m-with-exec(s,*boolean*) : active/désactive l'exécution des actions musicales de s,
— m-objects(s) : récupère la liste d'objets contenus dans s,
— m-group-objects(s,n) : récupère la liste d'objets contenus dans le groupe n de s,
— m-flush(s) : efface le contenu de s.

L'opérateur m-with-exec permet de désactiver les actions musicales, ce qui revient à supprimer la connexion du moteur de répartition à la sortie (voir figure 3.2). De cette manière, seules les actions a_c déclenchant des calculs sont exécutées. Ce mode d'opération a été conçu afin de composer des partitions dont le but est de calculer des structures en temps-réel à destination d'un interprète : par exemple la construction à la volée de partitions dynamiques.

Ces différents opérateurs et accesseurs peuvent être exécutés au travers d'interactions utilisateurs, par l'environnement extérieur (typiquement par un message de contrôle envoyé par un autre logiciel), mais également être exécutés automatiquement par la restitution d'objets s'ils sont inclus dans des tâches.

6.5.2 *Patch de contrôle*

Une première extension de la maquette d'OpenMusic a été proposée par Bresson et Agon (2006), en y intégrant un *patch de synthèse* responsable du calcul de la structure musicale équivalente à l'agencement du contenu de la maquette (c'est à dire la collecte des résultats des processus pour construire l'objet global équivalent — par exemple, pour créer un fichier son à partir d'un ensemble de paramètres de synthèse organisés dans le temps).

Nous associons à chaque meta-partition un *patch de contrôle* qui étend cette idée, permettant un accès à la structure et au contenu de la meta-partition. Celui-ci est évalué préalablement, et reste actif pendant la restitution. Le mécanisme de restitution étant dynamique grâce au système d'ordonnancement, toute opération effectuée par le patch de contrôle se répercute en temps-réel sur son contenu et en affecte éventuellement la restitution.

La figure 6.9 présente un patch de contrôle contenant un programme visuel qui accède aux objets de la meta-partition m grâce à l'accesseur m-objects. Il en extrait le premier élément, et un appel à m-move (a) affecte donc la position temporelle du premier objet du contenu de m (b).

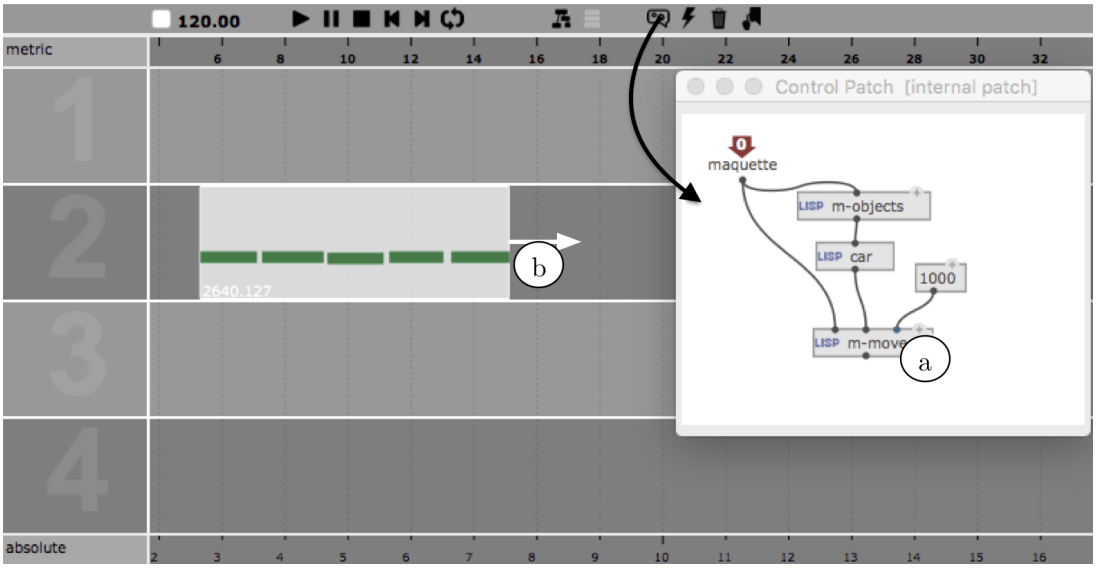


Figure 6.9 – Le patch de contrôle associé à une meta-partition : l'évaluation de son contenu peut affecter celui de la meta-partition. Ici, l'évaluation de la boîte m-move déplace le premier objet de la meta-partition de 1000 ms. La boîte *maquette* est une entrée spéciale permettant l'accès à la structure de la meta-partition.

6.5.3 ..... *Évaluations time/event-driven*

Le patch de contrôle permet donc de constituer un programme visuel dont les évaluations peuvent affecter le contenu de la meta-partition. Bien que ces évaluations puissent être déclenchées manuellement (*demand-driven*), nous fournissons des outils permettant de les rendre *time-driven* et *event-driven* (voir section 1.2.1). Pour cela, nous introduisons différents objets dans cette section.

Serveurs asynchrones. OpenMusic fournit des boîtes fonctionnelles implémentant des serveurs asynchrones pour la réception de messages. Par exemple, osc-receive permet la réception des messages OSC (Wright, 2005). La boîte osc-receive se comporte comme une boîte standard dont la valeur change à chaque réception d'un message OSC. Ainsi, en utilisant le comportement réactif d'OpenMusic (voir section 1.2.2), il est possible de propager tout message asynchrone en réception dans le programme visuel, et donc dans le patch de contrôle afin de modifier automatiquement le contenu de la meta-partition. Cela permet à l'environnement extérieur de contrôler des appels de fonctions affectant la meta-partition de manière asynchrone.

La figure 6.10 illustre l'utilisation d'un tel serveur : le contenu de la meta-partition peut être affecté sans interaction directe de l'utilisateur dans l'environnement.

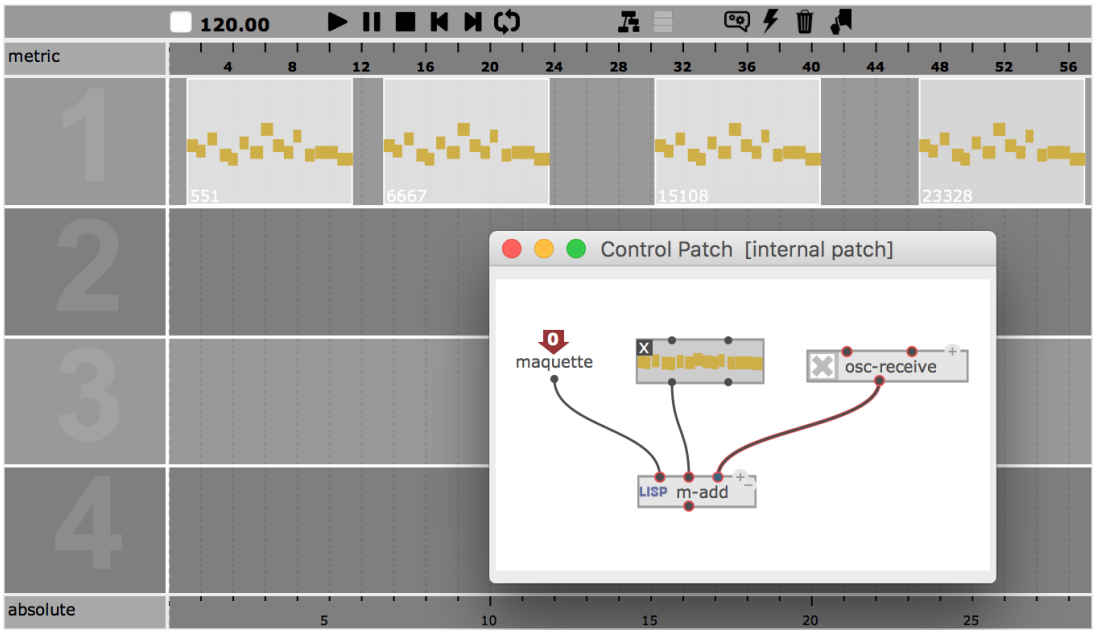


Figure 6.10 – Exemple d'automatisation du contrôle grâce à l'objet osc-receive. Chaque message reçu se propage automatiquement dans le programme visuel (connexions réactives en rouge) et déclenche un appel à m-add, qui ajoute alors une séquence de note prédéfinie au contenu de la meta-partition à une date donnée par le message.

Objets autonomes. Nous appelons objets autonomes les objets dont la restitution peut déclencher des fonctions définies par l'utilisateur. Parmi ces objets, nous pouvons citer l'automation ou encore la bpf (voir section 4.3) car ils exécutent des actions paramétrables. Nous introduisons également clock, un objet déclenchant périodiquement une fonction définie par l'utilisateur (par une abstraction), en lui fournissant comme paramètre un compteur qui s'incrémente à chaque période. L'utilisateur doit fournir une période temporelle p, et il a la possibilité de préciser une « durée de vie » d. En utilisation la formalisation introduite en section 3.1.3, le plan de cet objet est :

$$\forall t \in \mathbb{N},\ p_{\mathsf{clock}}(t) = \sum_{i=0}^{\lfloor \frac{d}{p} \rfloor} (p \times i, \lambda(i)).$$

Cet objet déclenche des évaluations périodiques dans le patch de contrôle, et permet ainsi d'automatiser l'évolution du contenu de la meta-partition. La figure 6.11 illustre l'utilisation d'un objet clock : le contenu de la meta-partition est affecté périodiquement, sans interaction utilisateur.

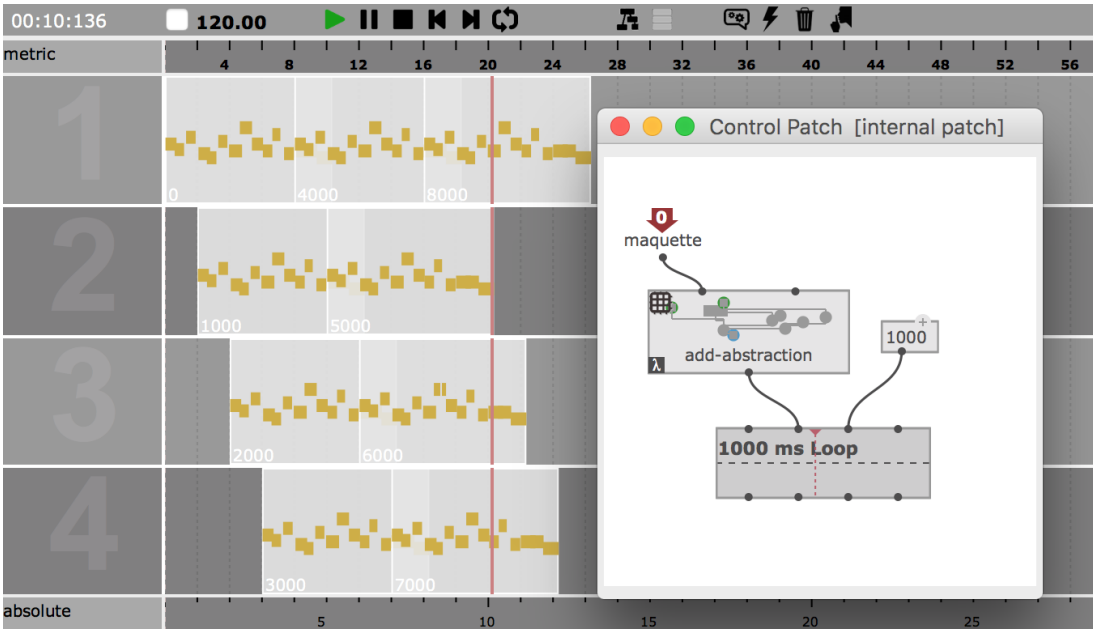


Figure 6.11 – Exemple d'automatisation du contrôle grâce à l'objet clock. Chaque période d'horloge déclenche un appel à la fonction définie par l'utilisateur et ajoute une séquence de note prédéfinie à la date $d = n \times 1000$, n étant le compteur interne à un objet clock.

interface. Finalement, nous définissons l'objet interface qui permet à l'utilisateur d'attribuer une interface graphique à certains éléments du patch de contrôle (boutons, *sliders*, entrées texte, *etc.*). La figure 6.12 illustre l'utilisation d'une interface : l'utilisateur peut utiliser le mécanisme d'évaluation des opérateurs de l'API de la meta-partition dans une interface graphique manipulable hors du patch de contrôle. Cet objet facilite le contrôle des patches complexes, en ne conservant que quelques paramètres visibles. [2]

6.6 Exemples

Cette section présente des applications qui illustrent des exemples de meta-composition par meta-partition, et qui font usage des outils présentés. [3]

2. Ce type de mécanisme est comparable au mode « présentation » de l'environnement Max, où l'utilisateur peut choisir quel objet doit rester visible lors du passage dans cette vue.

3. Les vidéos des exemples présentés en section 6.6.1 et 6.6.2 sont disponibles à l'adresse http://repmus.ircam.fr/efficace/wp/musical-processes.

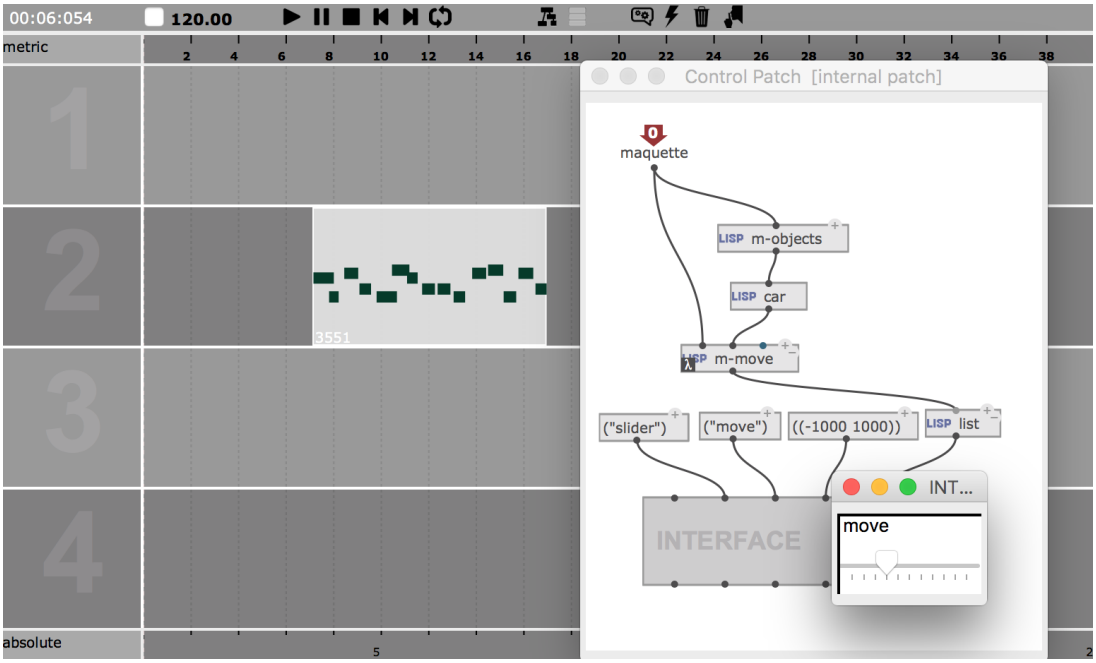


Figure 6.12 – Exemple de création d'une interface personnalisée. Ici, un appel à m-move est contrôlé par un curseur dont la valeur est passée en paramètre à l'appel (et en déclenche automatiquement l'évaluation à chaque modification).

6.6.1 ..... *Génération musicale dynamique avec spécifications formelles*

conception. Le premier exemple se place dans le cadre de systèmes de génération musicale automatique, alliant spécifications formelles de structures temporelles et interactivité. L'improvisation homme-machine (Assayag et al., 2006) par exemple, fait partie de cette catégorie de systèmes. L'objectif est ici d'intégrer des « agents » interactifs générateurs de structures musicales. Nous considérons pour cela le moteur de génération ImproteK (voir section 1.3.2), un système dédié à l'improvisation guidée homme-machine. Ce système génère des improvisations en guidant la navigation à travers une mémoire musicale grâce à un scénario (Nika, 2016). Il est construit sous la forme d'une chaîne de modules : un modèle de génération musicale guidée ; une architecture réactive qui gère la ré-écriture des anticipations musicales en réponse à des contrôles dynamiques (Nika et al., 2015) ; et des mécanismes de synchronisation pour adapter la restitution audio ou MIDI à une pulsation non-métronomique pendant la performance. La figure 6.13 décrit une possible intégration de ce moteur de génération dans un contexte de meta-cao.

Nous considérons deux agents d'improvisation (*voice 1* et *voice 2*), jouant de courtes phrases musicales (solos) tour à tour (sur le modèle de jeu « questions/réponses »)

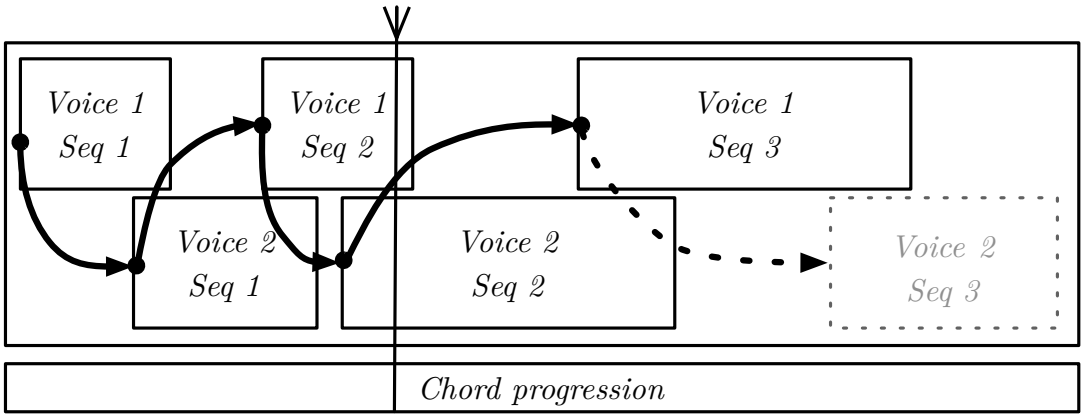


Figure 6.13 – Deux agents générant à la volée des questions/réponses suivant une grille d'accords prédéfinie.

suivant une grille d'accords prédéfinie (*chord progression*). Ces solos se chevauchent et ont des durées variables. L'idée portée par cette intégration est que chaque phrase musicale générée est déterminée par la grille d'accords (le « scénario ») mais également par des paramètres contextuels du passé proche. Les agents ont connaissance l'un de l'autre afin que le début de chaque solo poursuive de manière pertinente la fin du précédent en assurant une continuité musicale. En d'autres termes, chaque séquence de notes jouée par un agent doit influencer et prévoir dynamiquement la prochaine séquence jouée par l'autre agent.

réalisation. Chaque agent décrit précédemment est implémenté dans un patch OpenMusic (voir figure 6.14). Intégré dans une meta-partition, ce patch (ou processus compositionnel) est une abstraction dynamique : il sera exécuté par le moteur de calcul quand la date de restitution atteint sa position. Ce calcul procède à deux opérations :

1. Il génère des données (une séquence MIDI) suivant le scénario (grille d'accords) et le solo généré précédemment par l'autre agent ;
2. Il ajoute un autre agent sur l'autre voix, via la fonction m-add.

Les deux agents sont globalement identiques, bien qu'ils fassent appel à deux instances distinctes du moteur de génération, avec des paramètres différents (afin de simuler deux musiciens différents).

La figure 6.15 montre l'interface principale de la meta-partition et la figure 6.16 le patch de contrôle associé. L'évaluation préliminaire du patch de contrôle construit deux instances du moteur ImproteK, les inclue dans les deux agents interconnectés, et insère le premier agent sur la piste 1. La piste d'accompagnement est pré-calculée (en utilisant une troisième instance du moteur génératif). Le reste du processus se déroule

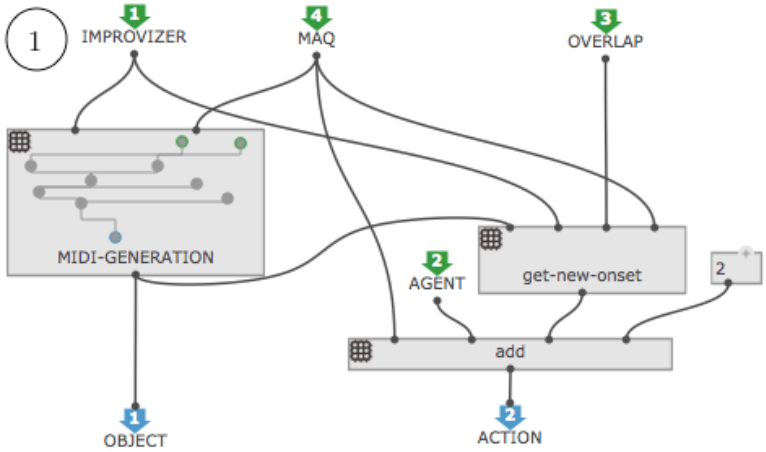


Figure 6.14 – Agent d'improvisation, créé dans le patch de contrôle (figure 6.16). Génère une séquence MIDI (*ouput* 1) et prévoit la génération d'un autre agent à la fin de la séquence (*ouput* 2).

ensuite automatiquement à la restitution, calculant les séquences alternativement grâce aux agents, qui s'instancient mutuellement tour à tour.

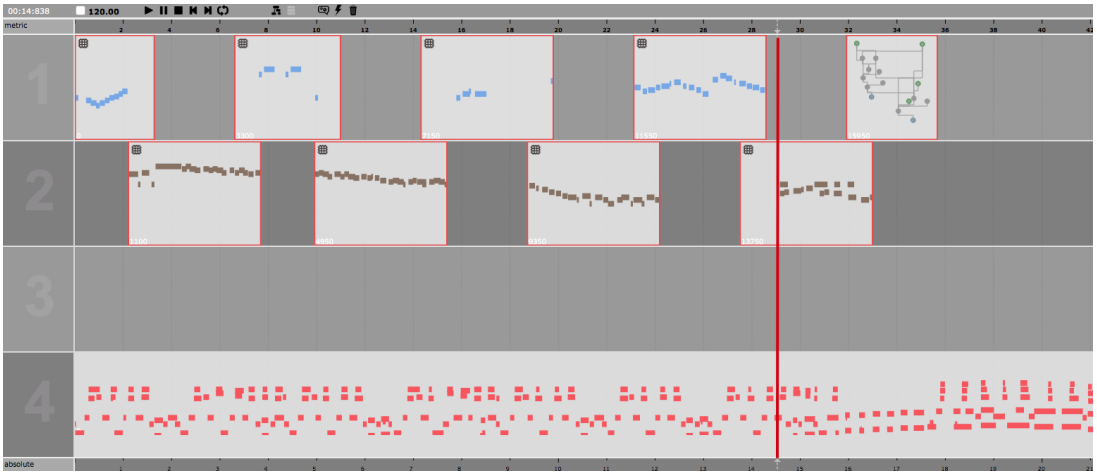


Figure 6.15 – Meta-partition en cours de restitution de l'exemple d'improvisation. Remarquons que le dernier agent est encore sous forme de processus compositionnel et n'a pas encore été calculé.

6.6.2 ..... *Contrôle de la synthèse pour l'oto-émission acoustique*

conception. Nous considérons ici une partition contrôlant un système de synthèse sonore externe en temps-réel. L'exemple est basé sur les recherches d'Alex Che-

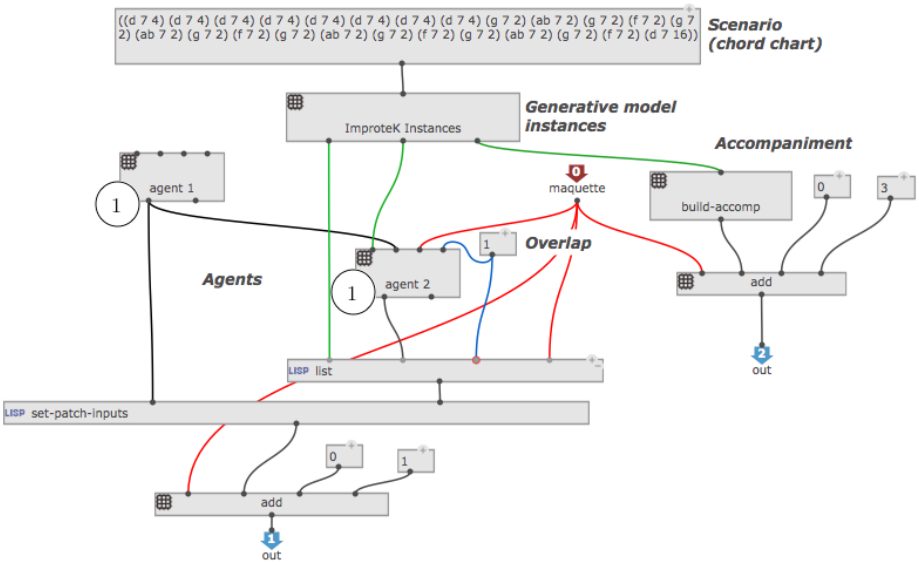


Figure 6.16 – Patch de contrôle de l'exemple d'improvisation. Des instances d'Improtek sont créées suivant une progression d'accords et intégrées dans les agents. Un agent et un accompagnement sont ajoutés à $t = 0$ dans la meta-partition. L'implémentation des agents est donnée en figure 6.14.

chile au CCRMA (Stanford University) sur les techniques des synthèse pour évoquer des oto-émissions acoustiques : sons générés dans l'oreille de l'auditeur à partir de combinaisons de fréquences pures (Chechile, 2015). Dans cette optique, des oscillateurs doivent être contrôlés simultanément à des fréquences précises, et avec des ratios spécifiques entre chaque voix.

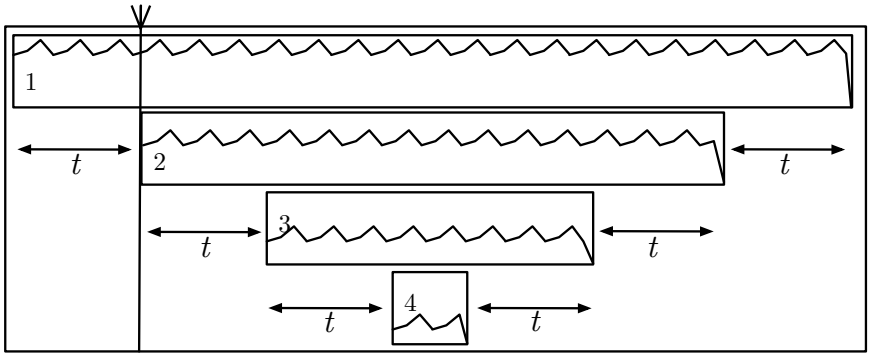


Figure 6.17 – Calcul et intégration de courbes de contrôle pour le contrôle de la synth!èse pour l'oto-émission acoustique. Les courbes 2 à 4 sont générées au temps t et déterminées par t et la courbe initiale. Les positions temporelles des courbes évoluent finement et continuellement durant la restitution.

La figure 6.17 montre l'allure globale de la meta-partition visée. Dans cet exemple, une courbe initiale (correspondant à une voix) est pré-calculée (en haut de la figure). L'utilisateur déclenche le calcul et l'intégration d'autres courbes de contrôle de manière telle que le résultat prend la forme d'une pyramide inversée : la durée d'une courbe dépend du temps à laquelle elle a été créée et intégrée à la partition. Pour induire le phénomène d'oto-émission acoustique, chaque courbe ajoutée doit être précisément déterminée selon la courbe initiale, et selon le temps de la restitution (afin de respecter la phase et les ratios fréquentiels entre les différents oscillateurs). Finalement, chaque courbe oscille continuellement sur l'axe temporel, produisant des variations et décalages dans le temps de la structure résultante.

RÉALISATION. L'implémentation de cet exemple fait usage des opérateurs m-add et m-move de l'API, ainsi que des objets clock et interface. Dans le patch de contrôle (figure 6.18), un objet initial est calculé suivant une liste de fréquences, une durée, un tempo, un port UDP et une adresse afin qu'OpenMusic communique avec l'environnement Max, qui exécute un ensemble d'oscillateurs.

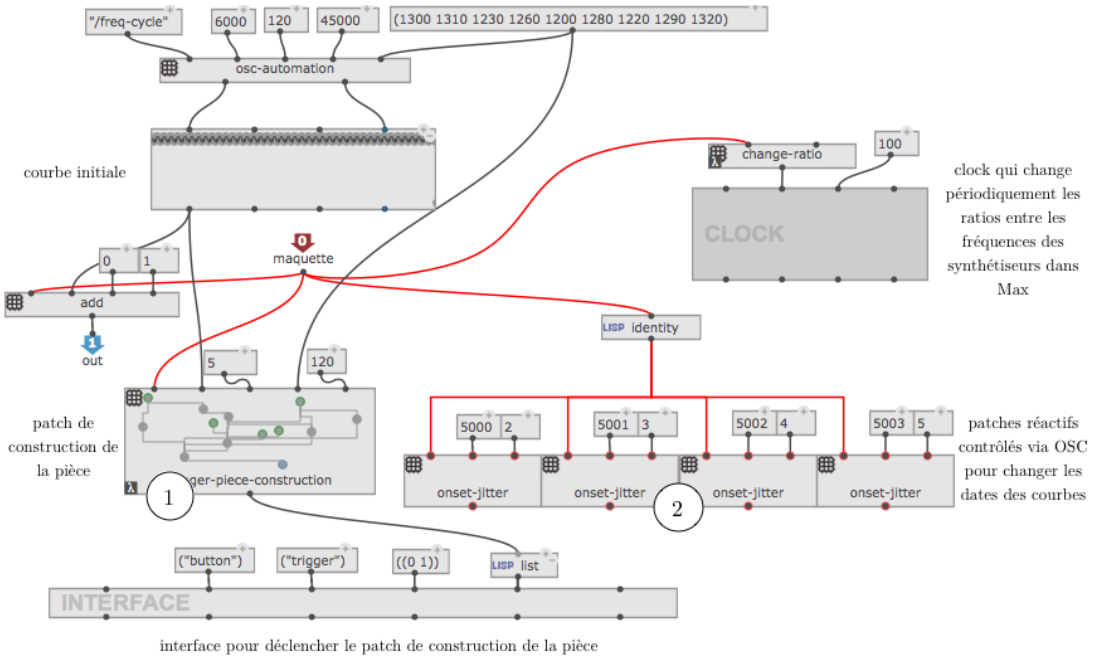


FIGURE 6.18 – Patch de contrôle pour la réalisation de l'exemple de synthèse pour l'oto-émission acoustique. Il contient : la courbe initiale qu'il ajoute dans la meta-partition à $t = 0$; le processus compositionnel construisant le reste de la meta-partition ; l'interface pour le déclencher ; des processus compositionnels réactifs de gigue (voir figure 6.20) ; un objet clock pour communiquer avec Max.

Une abstraction (*trigger-piece-construction*, visible en figure 6.18 et détaillée en figure 6.19) est mise en place afin de déclencher la construction et l'intégration des différentes courbes dans la meta-partition. Sur la figure 6.18, on voit que l'évaluation de cette abstraction est liée à une interface en permettant le déclenchement. Pendant la restitution, une interaction avec cette interface déclenche la génération et l'intégration des courbes additionnelles dans la meta-partition, chaque courbe étant un « multiple » de la courbe initiale. Les durées de ces courbes sont inversement proportionnelles à la durée écoulée entre le début de la restitution et l'interaction utilisateur, et est par conséquent décroissante en fonction du numéro de la courbe (voir figure 6.21).

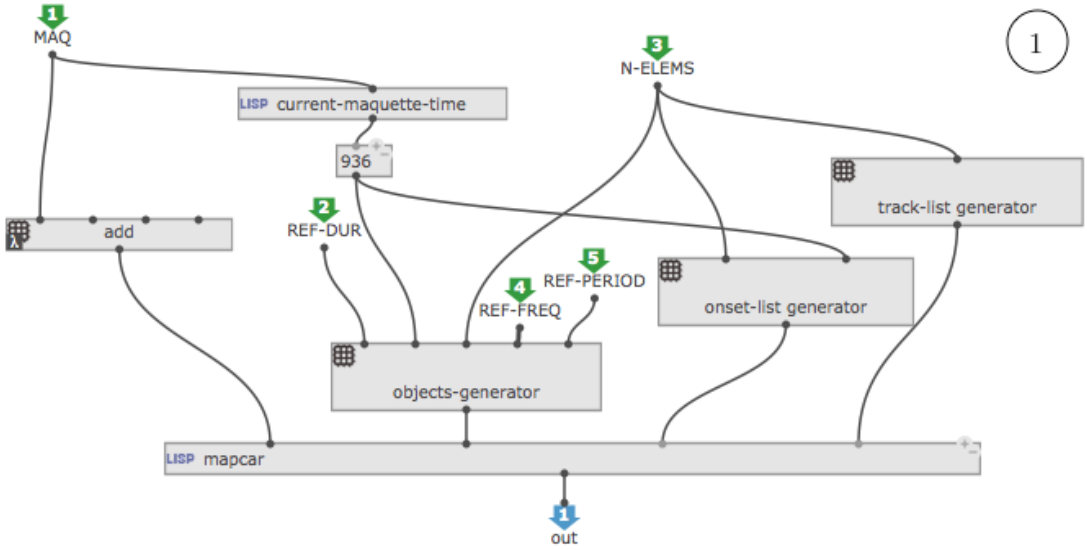


Figure 6.19 – Patch de construction des courbes de contrôle à partir de la courbe initiale et de la date courante de restitution de la meta-partition.

Grâce à OSC, les différentes courbes restitués par la meta-partition envoient des valeurs de fréquences aux oscillateurs de Max, et Max envoie des informations temporelles à OpenMusic. En plus des listes de fréquences envoyées à Max, l'objet clock examine périodiquement le nombre de pistes « actives » (dont un objet est en cours de restitution) de la meta-partition et envoie un message à Max déterminant le ratio entre les différentes fréquences des oscillateurs. Réciproquement, quatre flux de données continus de Max vers OpenMusic modifient les positions des courbes positionnées en pistes 2 à 5 de la meta-partition, via l'utilisation de quatre agents similaires (voir figure 6.20).

L'effet se manifeste par un léger mouvement des courbes de contrôle de gauche à droite (modifications des dates), pendant la restitution. Les figures 6.18 et 6.21 montrent la meta-partition et le patch de contrôle après que la génération de cinq courbes ait été déclenchée.

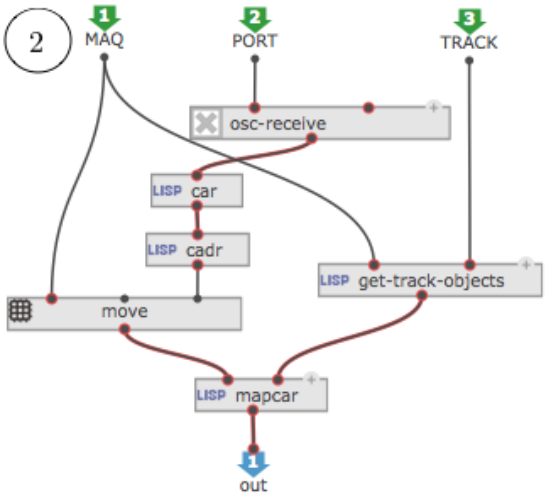


Figure 6.20 – Patch de l'agent réactif de gigue : modifie les dates des courbes de la meta-partition selon des messages OSC.

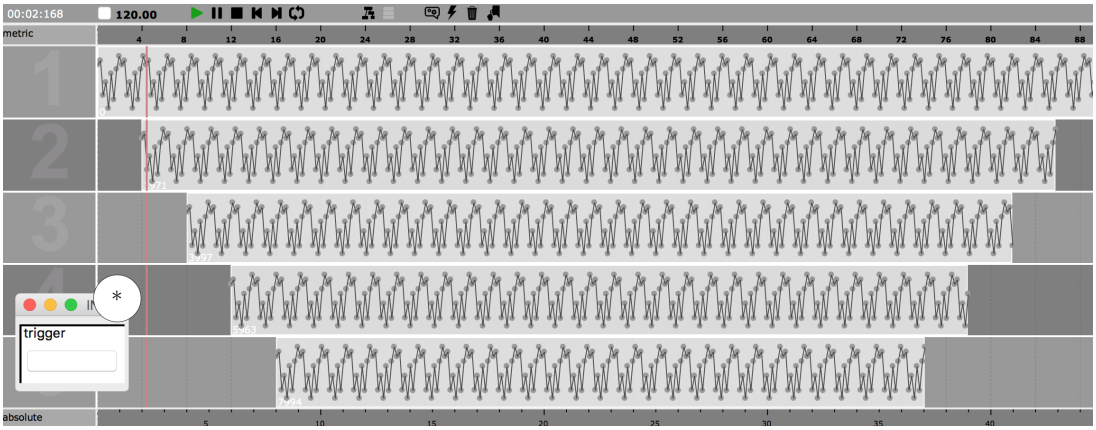


Figure 6.21 – Meta-partition de l'exemple de synthèse pour l'oto-émission acoustique en cours de restitution, suite au déclenchement de la constructions des courbes.
* : interface pour le déclenchement.

CONCLUSION

Les environnements dédiés à la composition musicale proposent des outils robustes pour la programmation et le calcul de structures musicales. Les environnement dédiés à la performance offrent des moyens de contrôler des processus génératifs en temps-réel. Nous avons exploré dans cette thèse une voie à l'intermédiaire de ces deux paradigmes.

La notion de meta-composition nous a permis de concevoir des programmes générant des données musicales pouvant être modifiées dynamiquement par des interactions ou par des actions générées par leur propre restitution. Notre objectif principal était la conception d'un système informatique fournissant des outils d'aide à la conception de tels programmes et capable de les exécuter. Pour cela, nous avons fait le choix de définir notre système comme une extension d'un langage « hors-temps » vers le domaine de l'interaction temps-réel.

CONTRIBUTIONS. Nous avons proposé une modélisation des données musicales de manière à intégrer à la fois les structures « apparentes » (objets, partitions) et les données d'exécution « bas-niveaux » qu'elles représentent (actions, dates, plans d'actions datées). Le système que nous avons conçu est basé sur une architecture composée de :

- Un moteur de calcul pour l'exécution asynchrone et parallèle de tâches ;
- Un moteur d'ordonnancement pour la construction de plans de restitution des données musicales ;
- Un moteur de répartition pour l'exécution dans le temps des actions des plans issus de l'ordonnancement.

Chaque opération d'un module peut entraîner une réaction dans les autres : la restitution peut déclencher des calculs, qui peuvent à leur tour venir influer sur l'ordonnancement et la répartition. Ce système peut être utilisé comme un « lecteur » élémentaire dans un environnement d'informatique musicale, tout en intégrant des capacités réactives. Il fait le pont entre le processus de calcul et le processus de restitution (la restitution des objets peut déclencher des calculs), s'adapte à la potentielle complexité de l'ordonnancement d'un objet et est robuste aux calculs et interactions erratiques venant modifier le contenu des objets (Bouche et al., 2016). Un module complémentaire vient s'insérer pour la gestion du signal audio et fait l'interface entre les caractéristiques asynchrones des processus compositionnels et les contraintes synchrones du rendu audio temps-réel.

Ce système reste avant tout un environnement de composition assistée par ordinateur permettant la composition de structure musicales complexes. Celles-ci peuvent désormais intégrer une dimension dynamique et être « hyper-connectées » à leur environnement.

PERSPECTIVES. Nos travaux se sont essentiellement focalisés sur l'ordonnancement d'actions dans un système musical. L'ordonnancement des tâches a été peu développé. Le calculateur introduit en section 3.3 implémente un algorithme simple : la file d'attente est triée par échéance croissante. Du fait du mécanisme d'anticipation introduit en section 6.3, les tâches peuvent arriver dans le calculateur dans un ordre différent de celui de leurs échéances. Cela ouvre la voix à l'implémentation d'algorithmes d'ordonnancement préemptifs, que nous n'avons pas exploré dans cette thèse, mais dont nous donnons des pistes en annexe C.

Notre système, à l'instar des environnements d'informatique musicale, est destiné à être déployé sur des machines « grand public ». Par conséquent, il est impossible de garantir certains scénarios, par exemple grâce à l'utilisation de WCET (voir section 2.3.2). Nous avons cependant envisagé une adaptivité en temps-réel des algorithmes d'ordonnancement et de restitution, en fonction des différentes perturbations auxquelles un système d'informatique musicale doit faire face. Nous donnons également des éléments concernant les performances et l'optimisation du système en annexe C.

Les travaux présentés dans ce manuscrit s'efforcent à présenter un système unifiant les aspects de gestions du temps, d'interaction, de calcul et de représentation musicale. Nous y parvenons, mais certains aspects restent limités : par exemple, la synchronisation d'actions dans le temps peut uniquement être réalisée dans les interfaces graphiques, mais n'est pas représentée dans notre modèle formel. En ce sens, nous envisageons le développement d'un modèle plus sophistiqué de la gestion des relations temporelles.

A

MODÈLE POUR LA PROGRAMMATION D'OBJETS

Dans cette annexe, nous présentons un modèle d'implémentation d'objets musicaux utilisant la classe s-object. Notre système étant implémenté dans l'environnement OpenMusic, le code est présenté dans le langage Common Lisp.

A.1 Définitions principales

```
;=== Object class definition ===
;inherits from the "s-object" class
(defclass MYOBJECT (s-object)
  [...])

;=== Object's duration getter ===
;returns the object's duration as an integer
(defmethod get-obj-dur ((self MYOBJECT-INSTANCE)) [...] )

;=== Actions collection ===
;returns a list of actions valid for [t1;t2[ as a list of lists
    ;format is ((date1 action1 dataset1) ... (dateN actionN datasetN))
        ;dates are unsigned integers
        ;actions are lambda functions
        ;datasets are lists
(defmethod collect-actions ((object MYOBJECT-INSTANCE) t1 t2 &optional parent)
 [...] )

;=== Tasks collection ===
;returns a list of tasks valid for [t1;t2[ as a list of lists
    ;format is ((date1 task1 dataset1) ... (dateN taskN datasetN))
        ;dates are unsigned integers
        ;tasks are lambda functions
        ;datasets are lists
(defmethod collect-tasks ((object MYOBJECT-INSTANCE) t1 t2 &optional parent)
 [...] )
```

A.2 Méthodes de transport

Les méthodes de transport peuvent être surchargées afin de transmettre les messages de contrôle lorsque la restitution d'un objet est réalisée, au moins partiellement, par d'autres applications. C'est par exemple le cas des objets ayant recours à un rendu audio temps-réel dans notre système.

```
(defmethod play ((object MYOBJECT) caller &key parent interval)
  ;Call the external "play" command
  [...])

(defmethod stop ((object MYOBJECT))
  ;Call the external "stop" command
  [...])

(defmethod pause ((object MYOBJECT))
  ;Call the external "pause" command
  [...])

(defmethod continue ((object MYOBJECT))
  ;Call the external "continue" command
  [...])

(defmethod jump-to ((self MYOBJECT) time)
  ;Call the external "jump-to" command
  [...])
```

B

DIAGRAMMES DE SÉQUENCE

Cette annexe présente les digrammes de séquence de la restitution des différents types de structures musicales. Nous rappelons que la méthode play qui déclenche la restitution peut être indifféremment appelée par le répartiteur ou par un utilisateur. Dans chacun de ces diagrammes, ne souhaitant pas faire d'hypothèse à ce sujet et par souci de simplification, l'appel de cette méthode n'est pas représenté sur une ligne de vie.

B.1 s-object « statique »

Nous appelons un s-object statique un objet dont la restitution ne déclenche pas d'action de calcul. Le calculateur n'apparait pas sur le diagramme, car les actions exécutées par le répartiteur sont uniquement destinées à la sortie du système.

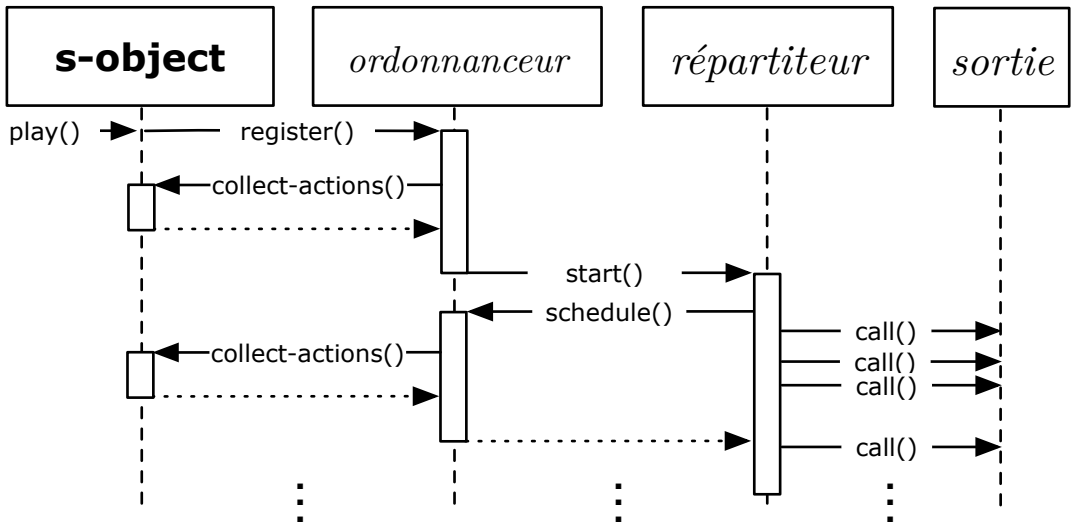


FIGURE B.1 – Diagramme de séquence de la restitution d'un s-object statique.

B.2 s-object « dynamique »

Un s-object dynamique est un objet dont la restitution déclenche des calculs. À la fin d'un calcul initié par le répartiteur, le calculateur notifie l'ordonnanceur de la complétion de la tâche afin de réaliser un ré-ordonnancement conditionnel (voir chapitre 3). Il existe trois comportements possibles (zone grisée en figure B.2) :

- L'ordonnanceur utilise les dates des tâches (appel de collect-tasks précédant l'appel à collect-actions) pour ajuster les horizons d'ordonnancement en conséquence. Par conséquent, si un calcul prend une $\Delta < \delta_{min}$, aucun ré-ordonnancement n'est nécessaire.
- Les horizons sont fixés par le programmeur, et le calcul termine pendant qu'un plan est en cours de restitution. L'ordonnanceur applique alors l'algorithme de ré-ordonnancement 2 vu en chapitre 3.
- Le calcul est déclenché par une interaction pendant qu'un plan est en cours de restitution. L'ordonnanceur applique alors l'algorithme de ré-ordonnancement 2 vu en chapitre 3.

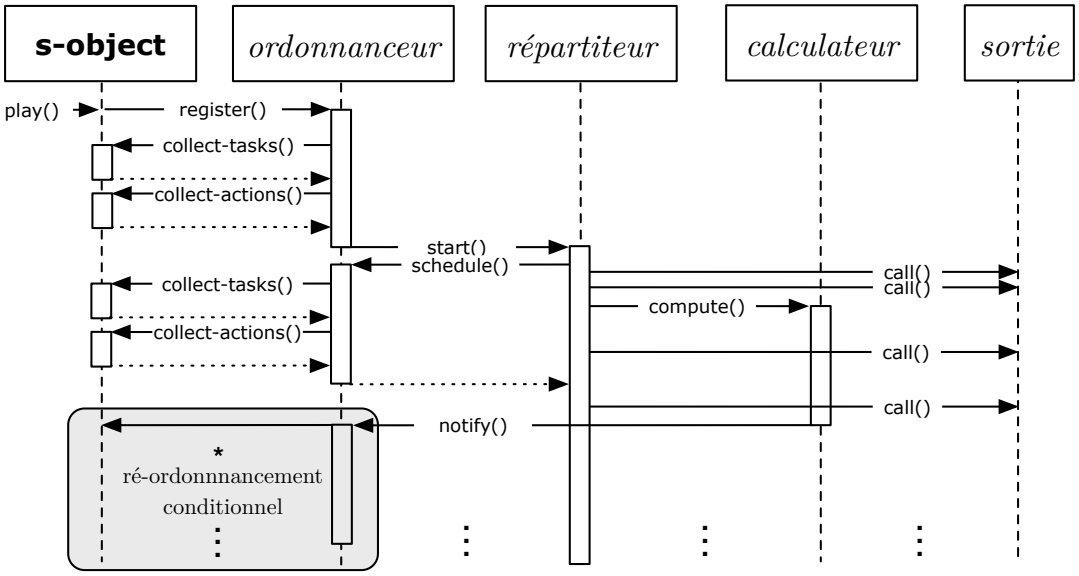


FIGURE B.2 – Diagramme de séquence de la restitution d'un s-object dynamique.
* : se référer au ré-ordonnancement conditionnel et à la figure 3.5 au chapitre 3.

B.3 Objet sound

Dans le diagramme de séquence de l'objet sound, le calculateur, répartiteur et ordonnanceur n'apparaissent pas. Bien qu'il y ait une communication avec l'ordonnanceur au démarrage de la restitution, celui-ci n'est plus actif dans la suite de la restitution et ne présente donc pas d'intérêt particulier.

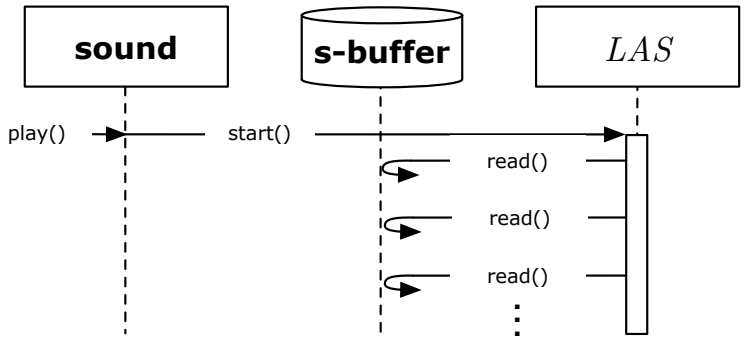


FIGURE B.3 – Diagramme de séquence de la restitution d'un objet sound.

B.4 Objet iae

La figure B.4 présente un diagramme de séquence de la restitution d'un objet iae dans notre système. On peut observer que les calculs sont de durées variables et viennent écrire dans le s-buffer de manière apériodique. L'évolution des intervalles d'ordonnancement n'est pas représentée. Dans le cas présenté, le premier calcul s'effectue à $t > 0$, car il n'y a pas de synthèse à opérer dès le début de la restitution de l'objet. Dans le cas où un calcul devrait être terminé à $t = 0$, la fonction collect-tasks peut renvoyer des tâches dont la date est négative : le système déclenche alors tous les calculs de date négative et attend leurs résultats avant de lancer la restitution (voir le cas présenté en B.5).

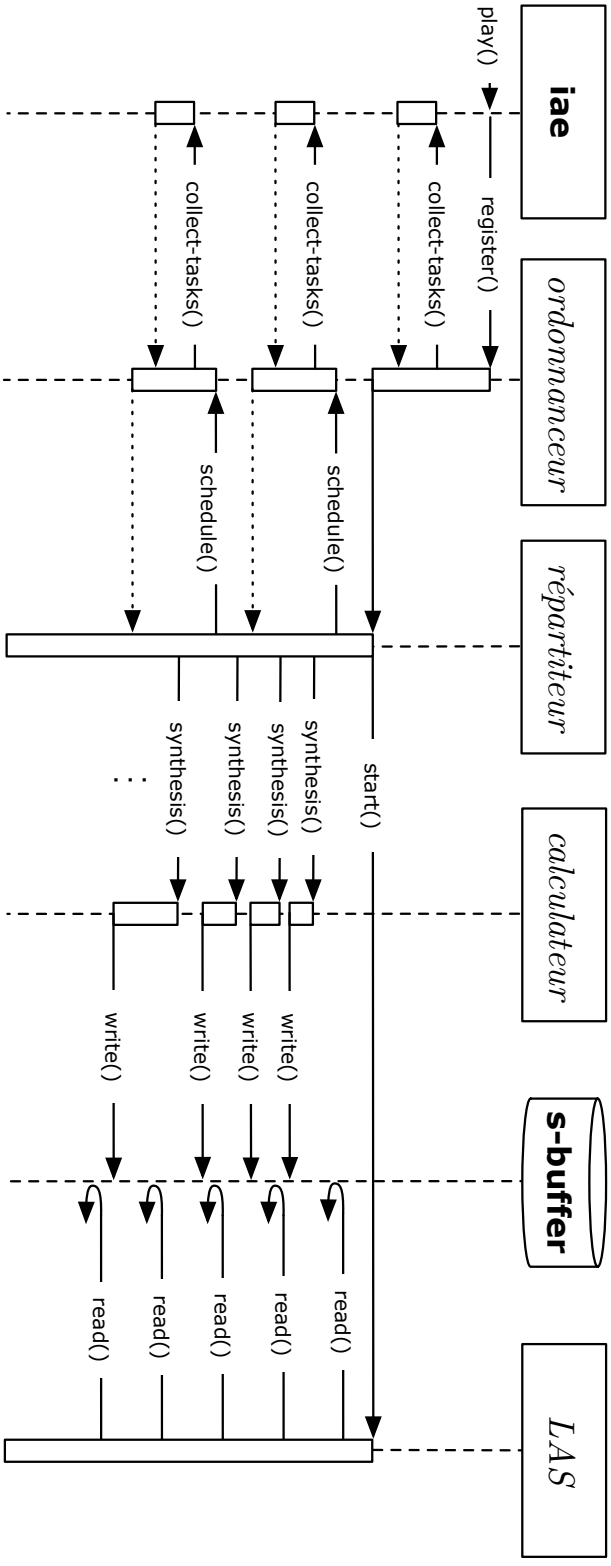


FIGURE B.4 – Diagramme de séquence du mécanisme de restitution de l'objet iae.

B.5 Objet spat-scene

La figure B.5 présente un diagramme de séquence de la restitution d'un objet spat-scene. On peut observer que les calculs sont cette fois de durées équivalentes et viennent écrire dans le s-buffer de manière périodique. Les intervalles d'ordonnancement n'évoluent pas. L'objet spat-scene déclenche un calcul à chaque début d'intervalle d'ordonnancement. Afin d'écrire les premiers échantillons avant le démarrage de la restitution, la fonction collect-tasks renvoie des tâches de date négative. De cette manière, la restitution démarre seulement une fois les premiers calculs effectués, et les calculs suivants seront toujours effectués avec un temps d'avance.

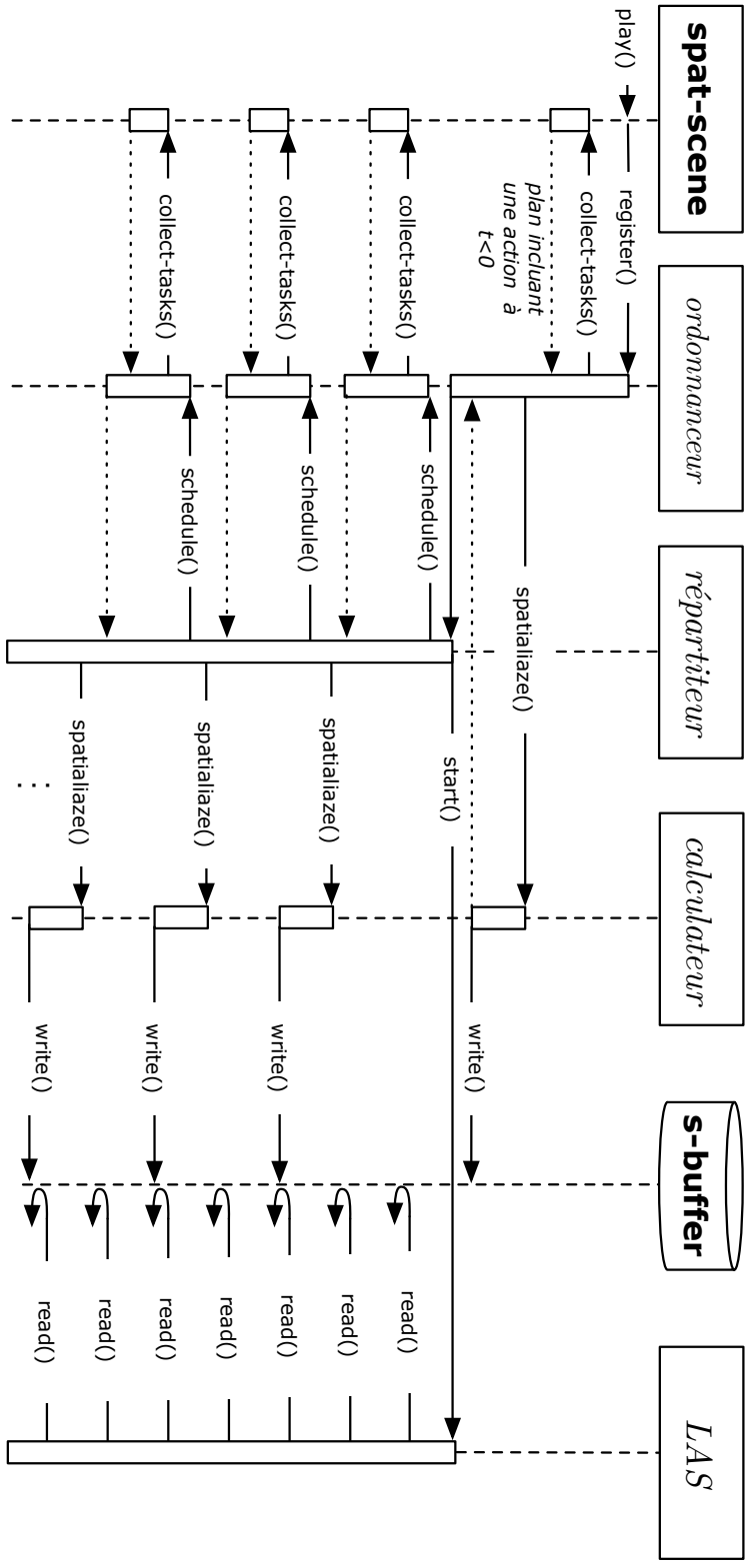


FIGURE B.5 – Diagramme de séquence du mécanisme de restitution de l'objet spat-scene.

C

PERFORMANCES & TIMING

Dans cette annexe, nous présentons différents points relatifs à l'optimisation de notre système. Tous les tests et mesures ont été réalisés sur un MacBook Pro 2,8GHZ Intel Core i7 (quatre cœurs) avec 16Go de mémoire vive, alors qu'un navigateur internet était en cours d'utilisation, afin d'accroître les perturbations. [1]

C.1 Validation empirique de l'instantanéité

Comme énoncé en chapitre 4, il peut être préférable pour le programmeur de vérifier si la durée des actions peut être considérée comme négligeable. Cela permet de garantir l'absence de retard dans le mécanisme de restitution. Nous donnons dans l'exemple ci-dessous une méthode simple de validation pour l'opération élémentaire d'addition, ainsi que pour l'envoi d'un événement MIDI.

```
;Addition simple
(defun addition ()
  (+ 1 2))
;Un million d'additions
(defun addition-loop ()
  (dotimes (i 1000000)
    (+ 1 2)))

;Une action MIDI
(defun midi-action ()
  (midi-send-evt *evt*))
;64 actions MIDI
(defun midi-action-loop ()
  (dotimes (i 64)
    (midi-send-evt *evt*)))
```

```
(time (addition))
> User time    =        0.000
> System time  =        0.000
> Elapsed time =        0.000

(time (addition-loop))
> User time    =        0.001
> System time  =        0.000
> Elapsed time =        0.000

(time (midi-action))
> User time    =        0.000
> System time  =        0.000
> Elapsed time =        0.000

(time (midi-action-loop))
> User time    =        0.001
> System time  =        0.000
> Elapsed time =        0.001
```

Nous pouvons observer que ces deux actions peuvent être considérées comme instantanées par notre système, sur la machine d'implantation. En effet, les temps d'exécutions de multiples actions reste inférieur ou égal à la granularité du système (1 ms).

1. Ces questions ont été étudiées notamment au cours du stage de Samuel Bell-Bell, étudiant en Master 2 ATIAM – UPMC, co-encadré avec Jean–Louis Giavitto (2016).

C.2 Optimisation du moteur de calcul

C.2.1 *Parallélisation et dépendance des tâches*

Nous avons vu en section 3.3 que le moteur de calcul peut être réalisé comme un processus unique ou comme un ensemble de processus. Dans notre implémentation, nous utilisons un ensemble de processus sous la forme d'une *thread-pool*. Cela permet de profiter de la parallélisation des calculs quand cela est possible (c'est-à-dire quand les tâches sont indépendantes). Cependant, le nombre optimal de processus dans cet ensemble dépend de la machine sur laquelle le système est implanté, ainsi que du contexte dans le lequel il est exécuté (par exemple en fonction de la cohabitation avec différents programmes). Afin d'optimiser la parallélisation des calculs, notre système contient une routine (voir figure C.1) qui détermine empiriquement le nombre de threads optimaux en calculant la durée d'exécution totale d'un grand nombre de tâches arbitraires indépendantes, suivant différents nombre de processus dans la *thread-pool*.

Les ordinateurs grand public modernes ont des processeurs multi-cœurs. L'*hyper-threading* est une technologie implémentée par certains processeurs qui permet de simuler deux cœurs « virtuels » par cœur « physique ». En règle général, le nombre de *thread* optimal sera au maximum le nombre de cœurs virtuels. Cela tient au fait que pour un nombre de *thread* supérieur au nombre de cœurs virtuels, l'ordonnancement des processus par le système d'exploitation réalise de trop nombreux changements de contextes, essayant d'allouer des ressources équitables aux différents processus, et donc produisant de trop nombreuses interruptions.

La figure C.2 montre l'évolution des temps d'exécution de la routine en fonctions de nombre de *threads* exécutés. Sur notre machine de test (un processeur à 4 cœurs physiques et, du fait de l'*hyper-threading*, 8 virtuels), et dans le contexte dans lequel la routine est exécutée, le nombre de *threads* optimal est 3.

Cette routine peut être exécutée par l'utilisateur afin de paramétrer notre système automatiquement.

La parallélisation des tâches permet donc, dans une certaine mesure, d'optimiser les temps de calcul. Cependant, dans un environnement comme OpenMusic, il est possible d'avoir à traiter des calculs dépendants. Dans un contexte *multi-thread*, ordonner la liste des tâches selon leurs dépendances n'est pas viable. Pour cela, nous avons implémenté des mécanisme d'attente et de *callbacks* au sein du moteur de calcul : une tâche peut être placée en attente d'un résultat, et automatiquement déclenchée une fois celui-ci disponible.

```
(defvar *routine-start* 0)
(defvar *routine-end* 0)

(defun routine-start ()
  (setq *routine-start* (get-internal-real-time)))

(defun routine-end ()
  (setq *routine-end* (get-internal-real-time)))

(defun routine-task ()
  (let ((l (loop for i from 1 to 250000 collect i)))
    (mapcar #'(lambda (a) (expt 2 (mod a 5000))) l)
    t))

(defun optimize-thread-pool ()
  (let ((thread-number-list '(1 2 3 4 5 6 7 8 16 32 64))
        (min-time (most-positive-fixnum))
        (tasks-list (append `(routine-start)
                            (make-list 1000 :initial-element 'routine-task)
                            `(routine-end)))
        thread-pool)
    (loop for thread-number in thread-number-list
          do
          (setq thread-pool (build-thread-pool thread-number))
          (add-multiple-tasks thread-pool tasks-list)
          (wait-for *routine-end*)
          (if (< (- *routine-end* *routine-start*) min-time)
              (setq min-time (- *routine-end* *routine-start*)
                    optimum thread-number))
          (setq *routine-start* nil
                *routine-end* nil)
          (abort-thread-pool thread-pool))
    (build-thread-pool optimum)))
```

FIGURE C.1 – Code d'optimisation automatique du nombre de *threads* du moteur de calcul (*thread-pool*).

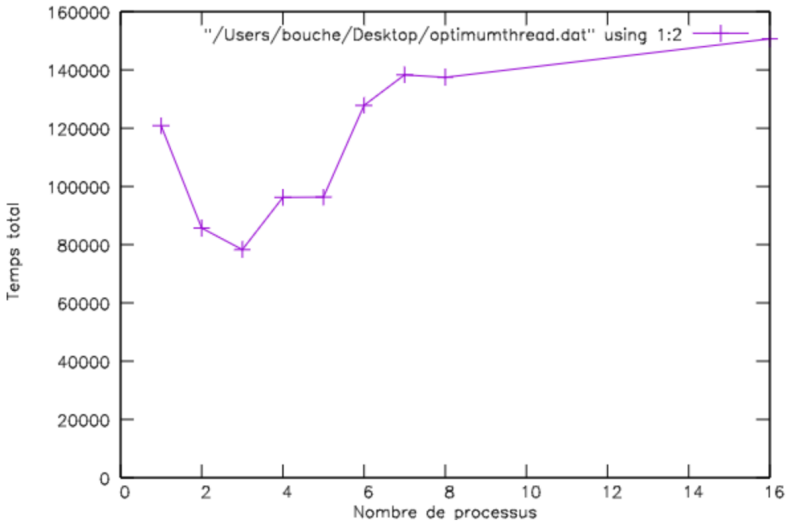


FIGURE C.2 – Détermination du nombre de *threads* optimal.

C.2.2 ..... *Distribution des temps de calcul*

L'utilisation d'un moteur de calcul *multi-thread* couplée au mécanisme d'anticipation vu en section 6.3 permet d'envisager un algorithme d'ordonnancement plus complexe qu'un simple tri, et également préemptif. Comme nous l'avons énoncé en section 2.3.4.1, l'utilisation de WCET n'est pas envisageable sur des machines grand public. Cependant, nous avons envisagé un « monitoring automatique » des temps de calcul pouvant conduire à l'élaboration d'ordonnancements optimaux. Un outil de mesure collecte diverses statistiques sur l'exécution des tâches, ainsi que sur le contexte dans lequel ces exécutions se déroulent. Des distributions des temps de calculs sont collectées et peuvent être analysées. La figure C.4 en donne un exemple.

Les fréquences des processeurs modernes varient au cours du temps, en fonction de nombreux paramètres (charge, température, optimisations, *etc.*). Nous avons remarqué que la pondération des temps de calcul des tâches par la fréquence du processeur leurs exécutions conduit à des distribution plus réduite. Cela permet d'imaginer un contrôle sur la fréquence du processeur à des fins d'optimisation.

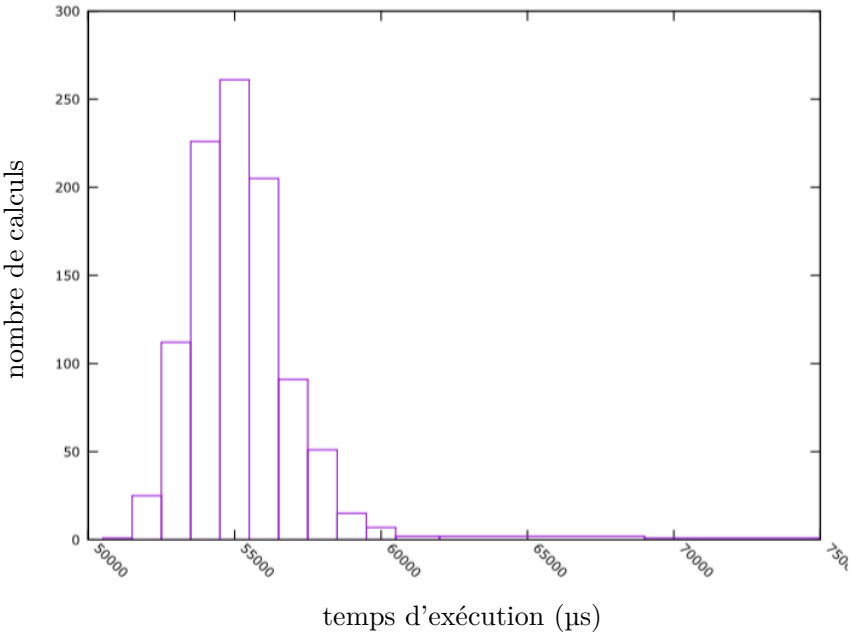


FIGURE C.3 – Exemple de distribution des temps de calcul pour une tâche arbitraire.

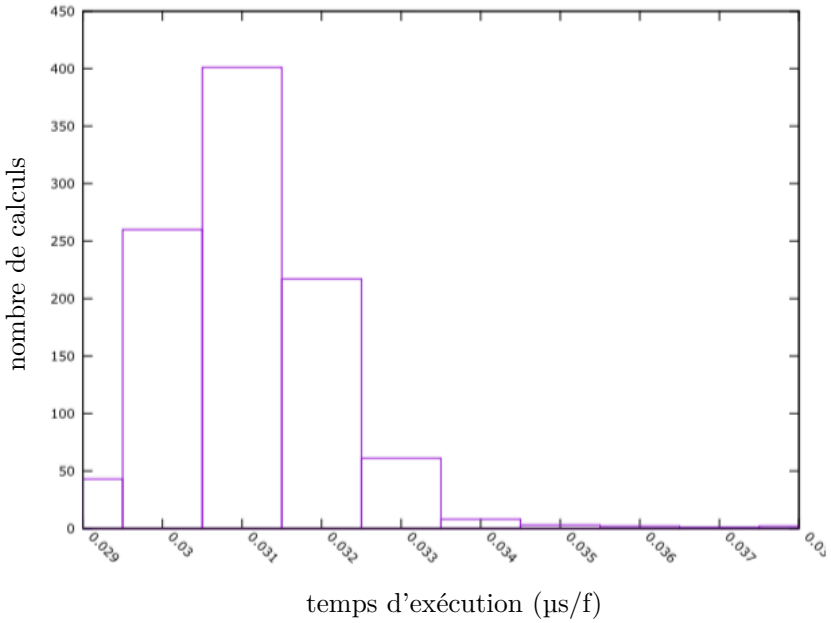


FIGURE C.4 – Distribution des temps de calcul de la figure C.4 pondérés par les fréquences du processeur.

Dans le système que nous avons présenté en chapitre 6, le déclenchement du calcul d'une abstraction peut être réalisé avec une avance déterminée par l'utilisateur (voir section 6.3). Lorsque des calculs sont trop complexes, ou lorsque qu'un partition en contient un trop grand nombre, il est possible que les résultats soient intégrés en retard par rapport à leur échéance (date de l'abstraction dans la partition). Pour cela, nous avons intégré un mécanisme de recensement des retards afin d'avertir l'utilisateur des retards détectés et de leurs valeurs.



FIGURE C.5 – Meta-partition contenant treize abstractions générant des séquences de notes arbitraires.

L'exemple de meta-partition présenté en figure C.5 contient treize abstractions. Leurs calculs seront déclenchés durant la restitution, aux dates définies par leur position horizontale. Après une restitution de la meta-partition, notre système fournit à l'utilisateur un historique des retards (voir figure C.6).

Grâce à cet historique des retards, l'utilisateur peut ajuster les *pre-delay* de chaque abstraction dynamique afin d'éviter des retards dans la prochaine exécution. De même, bien que nous n'ayons pas implémenté ce comportement à l'heure actuelle, ces anticipations pourraient être automatiquement appliquées par l'ordonnanceur. Dans la figure C.7, on peut voir un nouvel historique des retards après que l'utilisateur ait affecté des anticipations de 10 ms à chaque abstraction. Le système intègre à présent les résultats en avance.

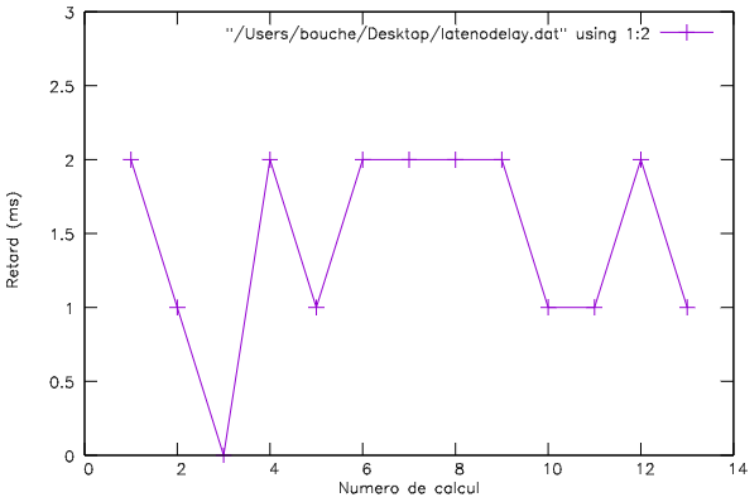


FIGURE C.6 – Retards d'intégration des résultats des treize calculs d'abstractions.

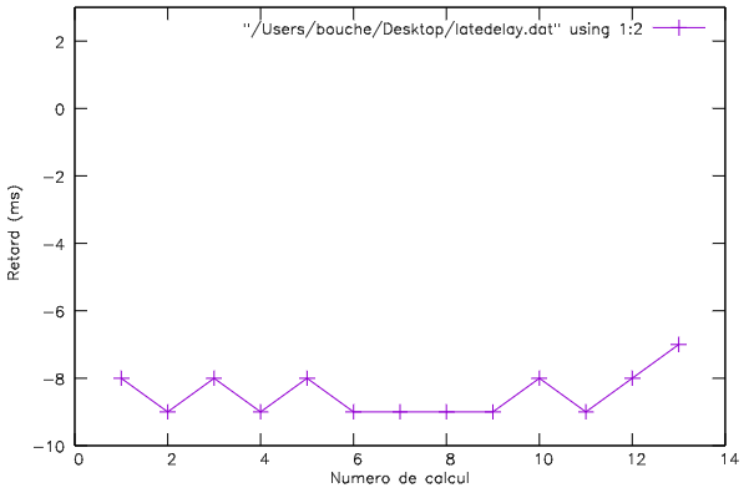


FIGURE C.7 – Retards d'intégration des résultats des treize calculs d'abstractions avec anticipation de 10 ms.

BIBLIOGRAPHIE

Carlos Agon. 2004, «Langages de programmation pour la composition musicale», Habilitation à Diriger des Recherches, Université Pierre et Marie Curie. (Cited on page 8.)

Andrea Agostini et Daniele Ghisi. 2013, «Real-Time Computer-Aided Composition with *bach*», *Contemporary Music Review*, vol. 32, n° 1, p. 41–48. (Cited on page 15.)

Andrea Agostini, Daniele Ghisi et Eric Maestri. 2016, «Recreating Gérard Grisey's Vortex Temporum with cage», dans *Proceedings of the International Computer Music Conference (ICMC)*, Utrecht, The Netherlands. (Cited on page 18.)

James F. Allen. 1983, «Maintaining Knowledge About Temporal Intervals», *Communications of the ACM*, vol. 26, n° 11, p. 832–843. (Cited on page 27.)

Sebastian Altmeyer, Sakthivel Manikandan Sundharam et Nicolas Navet. 2016, «The Case for FIFO Real-Time Scheduling», cahier de recherche, University of Luxembourg. (Cited on page 40.)

Pascalin Amagbégnon, Loïc Besnard et Paul Le Guernic. 1995, «Implementation of the Data-flow Synchronous Language SIGNAL», dans *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*, La Jolla, California, USA. (Cited on page 31.)

Jose A. Ambros-Ingerson et Sam Steel. 1988, «Integrating Planning, Execution and Monitoring», dans *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence (AAAI'88)*, Saint Paul, MN, USA. (Cited on page 34.)

Torsten Anders et Eduardo R. Miranda. 2011, «Constraint Programming Systems for Modeling Music Theories and Composition», *ACM Computer Surveys*, vol. 43, n° 4. (Cited on page 8.)

Gérard Assayag. 1998, «Computer Assisted Composition Today», dans *1st Symposium on Music and Computers*, Corfu, Greece. (Cited on page 7.)

Gérard Assayag, Georges Bloch, Marc Chemillier, Arshia Cont et Shlomo Dubnov. 2006, «Omax Brothers: A Dynamic Topology of Agents for Improvization Learning», dans *Workshop on Audio and Music Computing for Multimedia, ACM MultiMedia*, Santa Barbara, CA, USA. (Cited on page 97.)

Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon et Olivier Delerue. 1999, «Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic», *Computer Music Journal*, vol. 23, n° 3, p. 59–72. (Cited on page 8.)

Todd Austin, Eric Larson et Dan Ernst. 2002, «SimpleScalar: An Infrastructure for Computer System Modeling», *Computer*, vol. 35, n° 2, p. 59–67. (Cited on page 35.)

Pascal Baltazar, Théo de la Hogue et Myriam Desainte-Catherine. 2014, «i-score, an Interactive Sequencer for the Intermedia Arts», dans *Proceedings of the joint 40th International Computer Music Conference (ICMC) / 11th Sound and Music Computing Conference (SMC)*, Athens, Greece. (Cited on page 18.)

Marc Battier et Serge Lemouton. «Les étapes importantes de l'informatique musicale», Online: http://serge.lemouton.free.fr/paris8/chronologie.html. (Cited on page 5.)

Gérard Berry et Arshia Cont. 2010, «Le temps informatique : du temps réel au temps logique / La temporalité en informatique musicale : de la composition à la performance», Conférence enregistrée à l'IRCAM le 10/06/2010: http://medias.ircam.fr/xc4e939_le-temps-informatique-du-temps-reel-au-t. (Cited on page 28.)

Gérard Berry et Georges Gonthier. 1992, «The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation», *Science of Computer Programming*, vol. 19, n° 2, p. 87–152. (Cited on page 31.)

Anthony Beurivé. 2000, «Un logiciel de composition musicale combinant un modèle spectral, des structures hiérarchiques et des contraintes», dans *Actes des Journées d'Informatique Musicale*, Bordeaux, France. (Cited on page 27.)

Richard Bird et Philip Wadler. 1988, *Introduction to functional programming*, vol. 1, Prentice Hall New York. (Cited on page 7.)

Dimitri Bouche. 2013, «Dynamisation de l'architecture audio du logiciel OpenMusic», Rapport de stage – ENSEA / IRCAM. (Cited on page 66.)

Dimitri Bouche et Jean Bresson. 2015a, «Articulation dynamique de structures temporelles pour l'informatique musicale», dans *Modélisation des Systèmes Réactifs (MSR'15)*, Nancy, France. (Cited on page 39.)

Dimitri Bouche et Jean Bresson. 2015b, «Planning and Scheduling Actions in a Computer-Aided Music Composition System», dans *Proceedings of the ICAPS'15 Scheduling and Planning Applications woRKshop (SPARK) – International Conference on Automated Planning and Scheduling*, Jerusalem, Israel. (Cited on pages 39 and 43.)

Dimitri Bouche, Jean Bresson et Stéphane Letz. 2014, «Programmation and Control of Faust Sound Processing in OpenMusic», dans *Proceedings of the joint 40th International Computer Music conference (ICMC) / 11th Sound and Music Computing Conference (SMC)*, Athens, Greece. (Cited on pages 68 and 69.)

Dimitri Bouche, Jérôme Nika, Alex Chechile et Jean Bresson. 2016, «Computer-aided Composition of Musical Processes», *Journal of New Music Research*. (Cited on page 105.)

Craig Boutilier, Thomas Dean et Steve Hanks. 1999, «Decision-Theoretic Planning: Structural Assumptions and Computational Leverage», *Journal of Artificial Intelligence Research*, vol. 11, p. 1–94. (Cited on page 34.)

Jean Bresson. 2007, *La synthèse sonore en composition musicale assistée par ordinateur – Modélisation et écriture du son*, thèse de doctorat, Université Pierre et Marie Curie, Paris. (Cited on page 66.)

Jean Bresson. 2012, «Spatial Structures Programming for Music», dans *W21 Workshop on Spatial Computing Workshop (SCW) – International Conference on Autonomous Agents and Multiagent Systems (AAMAS'12)*, Valencia, Spain. (Cited on page 76.)

Jean Bresson et Carlos Agon. 2006, «Temporal Control over Sound Synthesis Processes», dans *Proceedings of the Sound and Music Computing conference (SMC)*, Marseille, France. (Cited on page 93.)

Jean Bresson, Carlos Agon et Gérard Assayag. 2009, «Visual Lisp/CLOS programming in OpenMusic», *Higher-Order and Symbolic Computation*, vol. 22, n° 1, p. 81–111. (Cited on page 10.)

Jean Bresson, Dimitri Bouche, Jérémie Garcia, Thibaut Carpentier, Florent Jacquemard, John MacCallum et Diemo Schwarz. 2015, «Projet EFFICACE : Développements et perspectives en composition assistée par ordinateur», dans *Actes des Journées d'Informatique Musicale*, Montréal, Canada. (Cited on page 2.)

Jean Bresson et Jean-Louis Giavitto. 2014, «A Reactive Extension of the OpenMusic Visual Programming Language», *Journal of Visual Languages and Computing*, vol. 25, n° 5, p. 363–375. (Cited on pages 12 and 79.)

Roland Cahen. 2012, «Topophonie Research Project», cahier de recherche, ENSCI – Les alteliers, Paris, France. (Cited on page 74.)

Roy H. Campbell et A. Nico Habermann. 1974, «The specification of process synchronization by path expressions», dans *Operating Systems (International Symposium)*,

Lecture Notes in Computer Science, vol. 16, édité par E. Gelenbe et C. Kaiser, Springer, p. 89–102. (Cited on page 32.)

Jonas Carlsson, Kent Palmkvist et Lars Wanhammar. 2006, «Synchronous Design Flow for Globally Asynchronous Locally Synchronous Systems», dans *Proceedings of the 10th WSEAS International Conference on Circuits (ICC'06)*, World Scientific and Engineering Academy and Society, Athens, Greece, p. 64–69. (Cited on page 68.)

Joel Chadabe. 1977, «Some Reflections on the Nature of the Landscape within Which Computer Music Systems are Designed», *Computer Music Journal*, vol. 1, n° 3, p. 5–11. (Cited on page 14.)

Joel Chadabe. 1984, «Interactive Composing: An Overview», *Computer Music Journal*, vol. 8, n° 1, p. 22–27. (Cited on page 17.)

Alex Chechile. 2015, «Creating spatial depth using distortion product otoacoustic emissions in music composition», dans *International Conference on Auditory Display (ICAD)*, Graz, Austria. (Cited on page 100.)

Philip R. Cohen et Hector J. Levesque. 1990, «Intention is Choice with Commitment», *Artificial Intelligence*, vol. 42, n° 2-3, p. 213–261. (Cited on page 34.)

Pierre Cointe. 1983, «Evaluation of Object oriented Programming from Simula to Smalltalk», dans *Proceedings of the Eleventh Simula User's Conference*, Paris, France. (Cited on page 18.)

Arshia Cont. 2008, «ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music», dans *Proceedings of the International Computer Music Conference (ICMC)*, Belfast, Ireland. (Cited on page 18.)

Graham Coulter-Smith. 2006, *Deconstructing Installation Art*, Online book: http://installationart.net/, CASIAD. (Cited on page 14.)

Myriam Desainte-Catherine et Antoine Allombert. 2004, «Specification of Temporal Relations Between Interactive Events», dans *Proceedings of the Sound and Music Computing conference (SMC)*, Paris, France. (Cited on page 27.)

Rajagopalan Desikan, Doug Burger et Stephen W. Keckler. 2001, «Measuring Experimental Error in Microprocessor Simulation», dans *Proceedings 28th Annual International Symposium on Computer Architecture*, Gothenburg, Sweden. (Cited on page 35.)

Marie E. desJardins, Edmund H. Durfee, Jr. Charles L. Ortiz et Michael J. Wolverton. 1999, «A Survey of Research in Distributed, Continual Planning», *AI Magazine*, vol. 20, n° 4. (Cited on pages 33 and 39.)

José Echeveste. 2015, *A programming language for Computer-Human Musical Interaction*, thèse de doctorat, Université Pierre et Marie Curie, Paris. (Cited on page 33.)

José Echeveste, Arshia Cont, Jean-Louis Giavitto et Florent Jacquemard. 2013, «Operational semantics of a domain specific language for real time musician-computer interaction», *Discrete Event Dynamic Systems*, vol. 23, n° 4, p. 343–383. (Cited on page 18.)

Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna et Antonio Mancina. 2009, «An implementation of the earliest deadline first algorithm in Linux», dans *Proceedings of the ACM symposium on Applied Computing*, Honolulu, Hawaii, USA, p. 1984–1989. (Cited on page 40.)

George Ferguson, James Allen et Brad Miller. 1999, «Mixed-Initiative Interaction», *IEEE Intelligent Systems*, vol. 14, n° 5, p. 14–23. (Cited on page 34.)

Dominique Fober, Yann Orlarey et Stephane Letz. 2011, «FAUST Architectures Design and OSC Support», dans *Proceedings of the International Conference on Digital Audio Effects (DAFx-11)*, Paris, France. (Cited on page 68.)

Jérémie Garcia, Thibaut Carpentier et Jean Bresson. 2016, «Interactive-Compositional Authoring of Sound Spatialization», *Journal of New Music Research*. (Cited on page 77.)

Ronald L. Graham, Eugene L. Lawler, Jan Karel Lenstra et AHG Rinnooy Kan. 1979, «Optimization and approximation in deterministic sequencing and scheduling: A survey», *Annals of Discrete Mathematics*, vol. 5, p. 287–326. (Cited on page 34.)

Nicholas Halbwachs, Paul Caspi, Pascal Raymond et Daniel Pilaud. 1991, «The synchronous data flow programming language LUSTRE», *Proceedings of the IEEE*, vol. 79, n° 9, p. 1305–1320. (Cited on page 31.)

Josué Hernandez et Jorge Torres. 2013, «Electromechanical design: Reactive planning and control with a mobile robot», dans *Proceedings of the 10th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, Mexico City, Mexico. (Cited on page 38.)

Lejaren Hiller. 1969, «Some Compositional Techniques Involving The Use of Computers», dans *Music by Computers*, édité par H. Von Foerster, John Wiley & Sons, p. 71–83. (Cited on page 7.)

Lejaren Hiller et Leonard Isaacson. 1959, *Experimental Music: Composition with an Electronic Computer*, McGraw-Hill, New York. (Cited on page 6.)

Thom Holmes. 2002, *Electronic and Experimental Music: Pioneers in Technology and Composition*, Routledge. (Cited on page 5.)

Paul Hudak, Tom Makucevich, Syam Gadde et Bo Whong. 1995, «Haskore Music Notation - An Algebra of Music», *Journal of Functional Programming*, vol. 6, n° 3, p. 465–483. (Cited on page 7.)

Marc Huou et Jean-Pierre Elloy. 1995, «Sémantique du parallélisme et du choix du langage Electre», *Informatique Théorique et Applications*, vol. 29, n° 4, p. 315–338. (Cited on page 32.)

Jean-Marc Jot. 1999, «Real-time Spatial Processing of Sounds for Music, Multimedia and Interactive Human-computer Interfaces», *Multimedia Systems*, vol. 7, n° 1, p. 55–69. (Cited on page 76.)

Jean-Marc Jot et Antoine Chaigne. 1991, «Digital delay networks for designing artificial reverberators», dans *AES 90th Convention*, Paris, France. (Cited on page 76.)

Jean-Pascal Jullien. 1995, «Structured Model for the Representation and the Control of Room Acoustical Quality», dans *International Congress on Acoustics (ICA)*, Trondheim, Norway. (Cited on page 76.)

Gilles Kahn. 1974, «The semantics of a simple language for parallel programming», dans *Information Processing: Proceedings of IFIP Congress 74*, Stockholm, Sweden, p. 471–475. (Cited on page 32.)

Mark Kahrs. 1993, «Dream Chip 1: A timed Priority Queue», *IEEE Micro*, vol. 13, n° 4, p. 49–51. (Cited on page 36.)

Ronald Kriemann. 2003, *Implementation and Usage of a Thread Pool based on POSIX Threads*, Max Planck Institute, Leipzig, Germany. (Cited on page 49.)

Chris Lattner et Vikram Adve. 2004, «LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation», dans *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California. (Cited on page 68.)

Mikael Laurson et Jacques Duthen. 1989, «Patchwork, a Graphic Language in PreForm», dans *Proceedings of the International Computer Music Conference (ICMC)*, Ohio State University, USA. (Cited on page 8.)

Mikael Laurson, Mika Kuuskankare et Vesa Norilo. 2009, «An Overview of PWGL, a Visual Programming Environment for Music», *Computer Music Journal*, vol. 33, n° 1, p. 19–31. (Cited on page 8.)

Jan van Leeuwen, éd. 1990, *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*, MIT Press. (Cited on page 80.)

Stephane Letz, Yann Orlarey et Dominique Fober. 1997, «L'environnement de composition musicale Elody», dans *Actes des Journées d'Informatique Musicale*, Lyon, France. (Cited on pages 8 and 10.)

Stéphane Letz, Yann Orlarey et Dominique Fober. 2014, «Spécification de l'extension libaudiostream.», cahier de recherche, Grame, Lyon, France. (Cited on page 67.)

Ruth Y. Litovsky, H. Steven Colburn, William A. Yost et Sandra J. Guzman. 1999, «The precedence effect», *The Journal of the Acoustical Society of America*, vol. 106, p. 1633–1654. (Cited on page 27.)

Philippe Manoury. 2008, «Considérations (toujours actuelles) sur l'état de la musique en temps réel», http://www.philippemanoury.com/?p=319. (Cited on page 33.)

Max Mathews. 1963, «The Digital Computer as a Musical Instrument», *Science*, vol. 142, n° 3592, p. 553–557. (Cited on page 6.)

Kenneth N. Mckay et Vincent C. Wiers. 2003, «Planning, Scheduling and Dispatching Tasks in Production Control», *Cognition, Technology & Work*, vol. 5, n° 2, p. 82–93. (Cited on page 36.)

James H. Morris. 1969, *Lambda-calculus models of programming languages*, thèse de doctorat, Massachusetts Institute of Technology. (Cited on page 7.)

Jérôme Nika. 2016, *Guiding human-computer music improvisation: introducing authoring and control with temporal scenarios*, thèse de doctorat, Université Pierre et Marie Curie, Paris. (Cited on page 97.)

Jérôme Nika, Dimitri Bouche, Jean Bresson, Marc Chemillier et Gérard Assayag. 2015, «Guided improvisation as dynamic calls to an offline model», dans *Proceedings of the Sound and Music Computing conference (SMC)*, Maynooth, Ireland. (Cited on pages ix and 97.)

Jérôme Nika, Marc Chemillier et Gérard Assayag. 2016, «ImproteK: Introducing scenarios into human-computer music improvisation», *ACM Computers in Entertainment, special issue on Musical Metacreation*. (Cited on page 17.)

Daniel Oppenheim. 1989, «DMIX: An Environment for Composition», dans *Proceedings of the International Computer Music Conference*, Ohio State University, USA. (Cited on page 8.)

Yann Orlarey. 2014, «Version librairie du compilateur Faust», cahier de recherche, Grame. https://hal.archives-ouvertes.fr/hal-00965271. (Cited on page 68.)

Yann Orlarey, Dominique Fober et Stephane Letz. 2009, «FAUST : an Efficient Functional Approach to DSP Programming», dans *New Computational Paradigms for Computer Music*, édité par G. Assayag et A. Gerzso, Editions Delatour France, p. 65–96. (Cited on page 68.)

Miller Puckette. 1988, «The Patcher», dans *Proceedings of the International Computer Music Conference (ICMC)*, Kologne, Germany. (Cited on page 15.)

Miller Puckette. 1991, «Combining Event and Signal Processing in the Max Graphical Programming Environment», *Computer Music Journal*, vol. 15, nº 3. (Cited on pages 8 and 15.)

Miller Puckette. 1996, «Pure Data: Another Integrated Computer Music Environment», dans *Proceedings of the Second Intercollege Computer Music Concerts*, Tachikawa, Japan. (Cited on pages 8 and 15.)

Miller Puckette. 2002, «Using Pd as a score language», dans *Proceedings of the International Computer Music Conference (ICMC)*, Göteborg, Sweden. (Cited on page 19.)

Miller Puckette. 2004, «A divide between 'compositional' and 'performative' aspects of Pd», dans *1st International Pd Convention*, Graz, Austria. (Cited on page 21.)

S. Ramesh, Sampada Sonalkar, Vijay D'silva, Naveen Chandra R. et B. Vijayalakshmi. 2004, «A Toolset for Modelling and Verification of GALS Systems», dans *Proceedings of the 16th International Conference on Computer Aided Verification*, *Lecture Notes in Computer Science*, vol. 3114, édité par R. Alur et D. A. Peled, Springer, p. 506–509. (Cited on page 68.)

Rasmus V. Rasmussen et Michael A. Trick. 2008, «Round robin scheduling – a survey», *European Journal of Operational Research*, vol. 188, nº 3, p. 617–636. (Cited on page 41.)

Xavier Rodet et Pierre Cointe. 1984, «FORMES: Composition and Scheduling of Processes», *Computer Music Journal*, vol. 8, nº 3, p. 32–50. (Cited on page 18.)

Luc Rondeleux. 1999, «Une histoire de l'informatique musicale entre macroforme et microcomposition», dans *Actes des Journées d'Informatique Musicale*. (Cited on page 5.)

Francesca Rossi, Peter van Beek et Toby Walsh, éd. 2006, *Handbook of Constraint Programming*, Elsevier. (Cited on page 8.)

Robert Rowe. 1996, «Incrementally improving interactive music systems», *Contemporary Music Review*, vol. 13, n° 2, p. 47–62. (Cited on page 17.)

James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy et William Lorensen. 1991, *Object-oriented modeling and design*, Prentice-Hall International. (Cited on page 7.)

Nicolas Scaringella, Yann Orlarey, Stephane Letz et Dominique Fober. 2003, «Automatic vectorization in faust», dans *Actes des Journées d'Informatique Musicale*, Montbeliard, France. (Cited on page 68.)

Diemo Schwarz. 2006, «Concatenative Sound Synthesis: The Early Years», *Journal of New Music Research*, vol. 35, n° 1, p. 3–22. (Cited on page 74.)

Marco Stroppa et Jacques Duthen. 1990, «Une représentation de structures temporelles par synchronisation de pivots», dans *Colloque Musique et Assistance Informatique*, Marseille, France. (Cited on page 90.)

Walid Talaboulma, Cristian Maxim, Adriana Gogonel, Yves Sorel et Liliana Cucu-Grosjean. 2015, «Estimation of probabilistic worst case execution time while accounting OS costs», dans *21st IEEE Real-Time Embedded Technology and Applications Symposium*, Seattle, United States. (Cited on page 37.)

Vincent Tiffon. 1994, *Recherches sur les musiques mixtes*, thèse de doctorat, Aix-Marseille Université. (Cited on page 14.)

Charlotte Truchet et Gérard Assayag, éd. 2011, *Constraint Programming in Music*, ISTE-Wiley. (Cited on page 8.)

Eric Cesar Jr. Vidal et Alexander Nareyek. 2011, «A Real-Time Concurrent Planning and Execution Framework for Automated Story Planning for Games», AAAI Technical Report – Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference. (Cited on page 39.)

Ge Wang. 2008, *The ChucK Audio Programming Language. A Strongly-timed and On-the-fly Environ/Mentality*, thèse de doctorat, Princeton University. (Cited on page 27.)

Ge Wang, Nicholas Bryan, Jieun Oh et Rob Hamilton. 2009, «Stanford Laptop Orchestra (SLOrk)», dans *Proceedings of the International Computer Music Conference (ICMC)*, Montreal, Canada. (Cited on page 14.)

Richard R. Weber, Pravin Varaiya et Jean Walrand. 1986, «Scheduling jobs with stochastically ordered processing times on parallel machines to minimize expected flowtime», *Journal of Applied Probability*, vol. 23, n° 3, p. 841–847. (Cited on page 41.)

Michael P. Wellman. 1992, «A General-Equilibrium Approach to Distributed Transportation Planning», dans *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, San Jose, USA, p. 282–289. (Cited on page 34.)

Reinhard Wilhelm. 2005, «Determining Bounds on Execution Times», dans *Embedded Systems Handbook*, édité par R. Zurawski, CRC Press. (Cited on page 35.)

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat et Per Stenström. 2008, «The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools», *ACM Transactions on Embedded Computing Systems*, vol. 7, n° 3. (Cited on page 35.)

Scott Wilson, David Cottle et Nick Collins. 2011, *The SuperCollider Book*, MIT Press. (Cited on page 8.)

M. Wright. 2005, «Open Sound Control: An enabling technology for musical networking», *Organised Sound*, vol. 10, n° 3. (Cited on page 94.)

Iannis Xenakis. 1992, *Formalized Music: Thought and Mathematics in Composition (revised)*, Harmonologia series, Pendragon Press. (Cited on page 6.)